Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Dive Into Python

20 May 2004

This book lives at http://diveintopython.org/. If you're reading it somewhere else, you may not have the latest version.

Chapter 1. Installing Python

Welcome to Python. Let's dive in. In this chapter, you'll install the version of Python that's right for you.

1.1. Which Python is right for you?

The first thing you need to do with Python is install it. Or do you?

If you're using an account on a hosted server, your ISP may have already installed Python. Most popular Linux distributions come with Python in the default installation. Mac OS X 10.2 and later includes a command−line version of Python, although you'll probably want to install a version that includes a more Mac−like graphical interface.

Windows does not come with any version of Python, but don't despair! There are several ways to point−and−click your way to Python on Windows.

As you can see already, Python runs on a great many operating systems. The full list includes Windows, Mac OS, Mac OS X, and all varieties of free UNIX−compatible systems like Linux. There are also versions that run on Sun Solaris, AS/400, Amiga, OS/2, BeOS, and a plethora of other platforms you've probably never even heard of.

What's more, Python programs written on one platform can, with a little care, run on *any* supported platform. For instance, I regularly develop Python programs on Windows and later deploy them on Linux.

So back to the question that started this section, "Which Python is right for you?" The answer is whichever one runs on the computer you already have.

1.2. Python on Windows

On Windows, you have a couple choices for installing Python.

ActiveState makes a Windows installer for Python called ActivePython, which includes a complete version of Python, an IDE with a Python−aware code editor, plus some Windows extensions for Python that allow complete access to Windows−specific services, APIs, and the Windows Registry.

ActivePython is freely downloadable, although it is not open source. It is the IDE I used to learn Python, and I recommend you try it unless you have a specific reason not to. One such reason might be that ActiveState is generally several months behind in updating their ActivePython installer when new version of Python are released. If you absolutely need the latest version of Python and ActivePython is still a version behind as you read this, you'll want to use the second option for installing Python on Windows.

The second option is the "official" Python installer, distributed by the people who develop Python itself. It is freely downloadable and open source, and it is always current with the latest version of Python.

Procedure 1.1. Option 1: Installing ActivePython

Here is the procedure for installing ActivePython:

1. Download ActivePython from http://www.activestate.com/Products/ActivePython/.
2. If you are using Windows 95, Windows 98, or Windows ME, you will also need to download and install Windows Installer 2.0 (http://download.microsoft.com/download/WindowsInstaller/Install/2.0/W9XMe/EN−US/InstMsiA.exe) before installing ActivePython.

3. Double−click the installer, `ActivePython−2.2.2−224−win32−ix86.msi`.
4. Step through the installer program.
5. If space is tight, you can do a custom installation and deselect the documentation, but I don't recommend this unless you absolutely can't spare the 14MB.
6. After the installation is complete, close the installer and choose Start−>Programs−>ActiveState ActivePython 2.2−>PythonWin IDE. You'll see something like the following:

```
PythonWin 2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)] on win32.
Portions Copyright 1994−2001 Mark Hammond (mhammond@skippinet.com.au) −
see 'Help/About PythonWin' for further copyright information.
>>>
```

Procedure 1.2. Option 2: Installing Python from Python.org (http://www.python.org/)

1. Download the latest Python Windows installer by going to http://www.python.org/ftp/python/ and selecting the highest version number listed, then downloading the `.exe` installer.
2. Double−click the installer, `Python−2.xxx.yyy.exe`. The name will depend on the version of Python available when you read this.
3. Step through the installer program.
4. If disk space is tight, you can deselect the HTMLHelp file, the utility scripts (`Tools/`), and/or the test suite (`Lib/test/`).
5. If you do not have administrative rights on your machine, you can select Advanced Options, then choose Non−Admin Install. This just affects where Registry entries and Start menu shortcuts are created.
6. After the installation is complete, close the installer and select Start−>Programs−>Python 2.3−>IDLE (Python GUI). You'll see something like the following:

```
Python 2.3.2 (#49, Oct  2 2003, 20:02:00) [MSC v.1200 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.0
>>>
```

1.3. Python on Mac OS X

On Mac OS X, you have two choices for installing Python: install it, or don't install it. You probably want to install it.

Mac OS X 10.2 and later comes with a command−line version of Python preinstalled. If you are comfortable with the command line, you can use this version for the first third of the book. However, the preinstalled version does not come with an XML parser, so when you get to the XML chapter, you'll need to install the full version.

Rather than using the preinstalled version, you'll probably want to install the latest version, which also comes with a graphical interactive shell.

Procedure 1.3. Running the Preinstalled Version of Python on Mac OS X

To use the preinstalled version of Python, follow these steps:

1. Open the `/Applications` folder.

2. Open the `Utilities` folder.
3. Double−click `Terminal` to open a terminal window and get to a command line.
4. Type **`python`** at the command prompt.

Try it out:

```
Welcome to Darwin!
[localhost:~] you% python
Python 2.2 (#1, 07/14/02, 23:25:09)
[GCC Apple cpp−precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you%
```

Procedure 1.4. Installing the Latest Version of Python on Mac OS X

Follow these steps to download and install the latest version of Python:

1. Download the `MacPython−OSX` disk image from http://homepages.cwi.nl/~jack/macpython/download.html.
2. If your browser has not already done so, double−click `MacPython−OSX−2.3−1.dmg` to mount the disk image on your desktop.
3. Double−click the installer, `MacPython−OSX.pkg`.
4. The installer will prompt you for your administrative username and password.
5. Step through the installer program.
6. After installation is complete, close the installer and open the `/Applications` folder.
7. Open the `MacPython−2.3` folder
8. Double−click `PythonIDE` to launch Python.

The MacPython IDE should display a splash screen, then take you to the interactive shell. If the interactive shell does not appear, select Window−>Python Interactive (**Cmd−0**). The opening window will look something like this:

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

Note that once you install the latest version, the pre−installed version is still present. If you are running scripts from the command line, you need to be aware which version of Python you are using.

Example 1.1. Two versions of Python

```
[localhost:~] you% python
Python 2.2 (#1, 07/14/02, 23:25:09)
[GCC Apple cpp−precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you% /usr/local/bin/python
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you%
```

1.4. Python on Mac OS 9

Mac OS 9 does not come with any version of Python, but installation is very simple, and there is only one choice.

Follow these steps to install Python on Mac OS 9:

1. Download the `MacPython23full.bin` file from http://homepages.cwi.nl/~jack/macpython/download.html.
2. If your browser does not decompress the file automatically, double−click `MacPython23full.bin` to decompress the file with Stuffit Expander.
3. Double−click the installer, `MacPython23full`.
4. Step through the installer program.
5. AFter installation is complete, close the installer and open the `/Applications` folder.
6. Open the `MacPython−OS9 2.3` folder.
7. Double−click `Python IDE` to launch Python.

The MacPython IDE should display a splash screen, and then take you to the interactive shell. If the interactive shell does not appear, select Window−>Python Interactive (**Cmd−0**). You'll see a screen like this:

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

1.5. Python on RedHat Linux

Installing under UNIX−compatible operating systems such as Linux is easy if you're willing to install a binary package. Pre−built binary packages are available for most popular Linux distributions. Or you can always compile from source.

Download the latest Python RPM by going to http://www.python.org/ftp/python/ and selecting the highest version number listed, then selecting the `rpms/` directory within that. Then download the RPM with the highest version number. You can install it with the **rpm** command, as shown here:

Example 1.2. Installing on RedHat Linux 9

```
localhost:~$ su −
Password: [enter your root password]
[root@localhost root]# wget http://python.org/ftp/python/2.3/rpms/redhat−9/python2.3−2.3−5pydotorg.i38
Resolving python.org... done.
Connecting to python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7,495,111 [application/octet−stream]
...
[root@localhost root]# rpm −Uvh python2.3−2.3−5pydotorg.i386.rpm
Preparing...                ########################################### [100%]
   1:python2.3              ########################################### [100%]
[root@localhost root]# python          ❶
Python 2.2.2 (#1, Feb 24 2003, 19:13:11)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2−4)] on linux2
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to exit]
[root@localhost root]# python2.3       ❷
```

```
Python 2.3 (#1, Sep 12 2003, 10:53:56)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to exit]
[root@localhost root]# which python2.3 ❸
/usr/bin/python2.3
```

❶ Whoops! Just typing **`python`** gives you the older version of Python −− the one that was installed by default. That's not the one you want.

❷ At the time of this writing, the newest version is called **`python2.3`**. You'll probably want to change the path on the first line of the sample scripts to point to the newer version.

❸ This is the complete path of the newer version of Python that you just installed. Use this on the `#!` line (the first line of each script) to ensure that scripts are running under the latest version of Python, and be sure to type **`python2.3`** to get into the interactive shell.

1.6. Python on Debian GNU/Linux

If you are lucky enough to be running Debian GNU/Linux, you install Python through the **apt** command.

Example 1.3. Installing on Debian GNU/Linux

```
localhost:~$ su -
Password: [enter your root password]
localhost:~# apt-get install python
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  python2.3
Suggested packages:
  python-tk python2.3-doc
The following NEW packages will be installed:
  python python2.3
0 upgraded, 2 newly installed, 0 to remove and 3 not upgraded.
Need to get 0B/2880kB of archives.
After unpacking 9351kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Selecting previously deselected package python2.3.
(Reading database ... 22848 files and directories currently installed.)
Unpacking python2.3 (from .../python2.3_2.3.1-1_i386.deb) ...
Selecting previously deselected package python.
Unpacking python (from .../python_2.3.1-1_all.deb) ...
Setting up python (2.3.1-1) ...
Setting up python2.3 (2.3.1-1) ...
Compiling python modules in /usr/lib/python2.3 ...
Compiling optimized python modules in /usr/lib/python2.3 ...
localhost:~# exit
logout
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> [press Ctrl+D to exit]
```

1.7. Python Installation from Source

If you prefer to build from source, you can download the Python source code from http://www.python.org/ftp/python/. Select the highest version number listed, download the `.tgz` file), and then do the usual **`configure`**, **`make`**, **`make`**

install dance.

Example 1.4. Installing from source

```
localhost:~$ su -
Password: [enter your root password]
localhost:~# wget http://www.python.org/ftp/python/2.3/Python-2.3.tgz
Resolving www.python.org... done.
Connecting to www.python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8,436,880 [application/x-tar]
...
localhost:~# tar xfz Python-2.3.tgz
localhost:~# cd Python-2.3
localhost:~/Python-2.3# ./configure
checking MACHDEP... linux2
checking EXTRAPLATDIR...
checking for --without-gcc... no
...
localhost:~/Python-2.3# make
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Modules/python.o Modules/python.c
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Parser/acceler.o Parser/acceler.c
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Parser/grammar1.o Parser/grammar1.c
...
localhost:~/Python-2.3# make install
/usr/bin/install -c python /usr/local/bin/python2.3
...
localhost:~/Python-2.3# exit
logout
localhost:~$ which python
/usr/local/bin/python
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
localhost:~$
```

1.8. The Interactive Shell

Now that you have Python installed, what's this interactive shell thing you're running?

It's like this: Python leads a double life. It's an interpreter for scripts that you can run from the command line or run like applications, by double−clicking the scripts. But it's also an interactive shell that can evaluate arbitrary statements and expressions. This is extremely useful for debugging, quick hacking, and testing. I even know some people who use the Python interactive shell in lieu of a calculator!

Launch the Python interactive shell in whatever way works on your platform, and let's dive in with the steps shown here:

Example 1.5. First Steps in the Interactive Shell

```
>>> 1 + 1                   ❶
```

```
2
>>> print 'hello world' ❷
hello world
>>> x = 1               ❸
>>> y = 2
>>> x + y
3
```

❶ The Python interactive shell can evaluate arbitrary Python expressions, including any basic arithmetic expression.

❷ The interactive shell can execute arbitrary Python statements, including the **print** statement.

❸ You can also assign values to variables, and the values will be remembered as long as the shell is open (but not any longer than that).

1.9. Summary

You should now have a version of Python installed that works for you.

Depending on your platform, you may have more than one version of Python intsalled. If so, you need to be aware of your paths. If simply typing **python** on the command line doesn't run the version of Python that you want to use, you may need to enter the full pathname of your preferred version.

Congratulations, and welcome to Python.

Chapter 2. Your First Python Program

You know how other books go on and on about programming fundamentals and finally work up to building a complete, working program? Let's skip all that.

2.1. Diving in

Here is a complete, working Python program.

It probably makes absolutely no sense to you. Don't worry about that, because you're going to dissect it line by line. But read through it first and see what, if anything, you can make of it.

Example 2.1. `odbchelper.py`

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

Now run this program and see what happens.

In the ActivePython IDE on Windows, you can run the Python program you're editing by choosing File−>Run... (**Ctrl−R**). Output is displayed in the interactive window.

In the Python IDE on Mac OS, you can run a Python program with Python−>Run window... (**Cmd−R**), but there is an important option you must set first. Open the `.py` file in the IDE, pop up the options menu by clicking the black triangle in the upper−right corner of the window, and make sure the Run as __main__ option is checked. This is a per−file setting, but you'll only need to do it once per file.

On UNIX−compatible systems (including Mac OS X), you can run a Python program from the command line: **`python odbchelper.py`**

The output of `odbchelper.py` will look like this:

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

2.2. Declaring Functions

Python has functions like most other languages, but it does not have separate header files like C++ or `interface`/`implementation` sections like Pascal. When you need a function, just declare it, like this:

```
def buildConnectionString(params):
```

Note that the keyword `def` starts the function declaration, followed by the function name, followed by the arguments in parentheses. Multiple arguments (not shown here) are separated with commas.

Also note that the function doesn't define a return datatype. Python functions do not specify the datatype of their return value; they don't even specify whether or not they return a value. In fact, every Python function returns a value; if the function ever executes a `return` statement, it will return that value, otherwise it will return `None`, the Python null value.

In Visual Basic, functions (that return a value) start with `function`, and subroutines (that do not return a value) start with `sub`. There are no subroutines in Python. Everything is a function, all functions return a value (even if it's `None`), and all functions start with `def`.

The argument, `params`, doesn't specify a datatype. In Python, variables are never explicitly typed. Python figures out what type a variable is and keeps track of it internally.

In Java, C++, and other statically−typed languages, you must specify the datatype of the function return value and each function argument. In Python, you never explicitly specify the datatype of anything. Based on what value you assign, Python keeps track of the datatype internally.

2.2.1. How Python's Datatypes Compare to Other Programming Languages

An erudite reader sent me this explanation of how Python compares to other programming languages:

statically typed language
: A language in which types are fixed at compile time. Most statically typed languages enforce this by requiring you to declare all variables with their datatypes before using them. Java and C are statically typed languages.

dynamically typed language
: A language in which types are discovered at execution time; the opposite of statically typed. VBScript and Python are dynamically typed, because they figure out what type a variable is when you first assign it a value.

strongly typed language
: A language in which types are always enforced. Java and Python are strongly typed. If you have an integer, you can't treat it like a string without explicitly converting it.

weakly typed language
: A language in which types may be ignored; the opposite of strongly typed. VBScript is weakly typed. In VBScript, you can concatenate the string `'12'` and the integer `3` to get the string `'123'`, then treat that as the integer `123`, all without any explicit conversion.

So Python is both *dynamically typed* (because it doesn't use explicit datatype declarations) and *strongly typed* (because once a variable has a datatype, it actually matters).

2.3. Documenting Functions

You can document a Python function by giving it a `doc string`.

Example 2.2. Defining the `buildConnectionString` Function's `doc string`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.
```

```
    Returns string."""
```

Triple quotes signify a multi−line string. Everything between the start and end quotes is part of a single string, including carriage returns and other quote characters. You can use them anywhere, but you'll see them most often used when defining a `doc string`.

Triple quotes are also an easy way to define a string with both single and double quotes, like `qq/.../` in Perl. Everything between the triple quotes is the function's `doc string`, which documents what the function does. A `doc string`, if it exists, must be the first thing defined in a function (that is, the first thing after the colon). You don't technically need to give your function a `doc string`, but you always should. I know you've heard this in every programming class you've ever taken, but Python gives you an added incentive: the `doc string` is available at runtime as an attribute of the function.

Many Python IDEs use the `doc string` to provide context−sensitive documentation, so that when you type a function name, its `doc string` appears as a tooltip. This can be incredibly helpful, but it's only as good as the `doc string`s you write.

Further Reading on Documenting Functions

- PEP 257 (http://www.python.org/peps/pep−0257.html) defines `doc string` conventions.
- *Python Style Guide* (http://www.python.org/doc/essays/styleguide.html) discusses how to write a good `doc string`.
- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses conventions for spacing in `doc strings` (http://www.python.org/doc/current/tut/node6.html#SECTION006750000000000000000).

2.4. Everything Is an Object

In case you missed it, I just said that Python functions have attributes, and that those attributes are available at runtime.

A function, like everything else in Python, is an object.

Open your favorite Python IDE and follow along:

Example 2.3. Accessing the `buildConnectionString` Function's `doc string`

```
>>> import odbchelper                                            ❶
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> print odbchelper.buildConnectionString(params) ❷
server=mpilgrim;uid=sa;database=master;pwd=secret
>>> print odbchelper.buildConnectionString.__doc__ ❸
Build a connection string from a dictionary

Returns string.
```

❶ The first line imports the `odbchelper` program as a module −− a chunk of code that you can use interactively, or from a larger Python program. (You'll see examples of multi−module Python programs in Chapter 4.) Once you import a module, you can reference any of its public functions, classes, or attributes. Modules can do this to access functionality in other modules, and you can do it in the IDE too. This is an important concept, and you'll talk more about it later.

❷ When you want to use functions defined in imported modules, you need to include the module name. So you can't just say `buildConnectionString`; it must be `odbchelper.buildConnectionString`. If you've used classes in Java, this should feel vaguely familiar.

❸ Instead of calling the function as you would expect to, you asked for one of the function's attributes, `__doc__`.

`import` in Python is like `require` in Perl. Once you `import` a Python module, you access its functions with *`module.function`*; once you `require` a Perl module, you access its functions with *`module`*`::`*`function`*.

2.4.1. The Import Search Path

Before you go any further, I want to briefly mention the library search path. Python looks in several places when you try to import a module. Specifically, it looks in all the directories defined in `sys.path`. This is just a list, and you can easily view it or modify it with standard list methods. (You'll learn more about lists later in this chapter.)

Example 2.4. Import Search Path

```
>>> import sys                 ❶
>>> sys.path                   ❷
['', '/usr/local/lib/python2.2', '/usr/local/lib/python2.2/plat-linux2',
'/usr/local/lib/python2.2/lib-dynload', '/usr/local/lib/python2.2/site-packages',
'/usr/local/lib/python2.2/site-packages/PIL', '/usr/local/lib/python2.2/site-packages/piddle']
>>> sys                        ❸
<module 'sys' (built-in)>
>>> sys.path.append('/my/new/path') ❹
```

❶ Importing the `sys` module makes all of its functions and attributes available.

❷ `sys.path` is a list of directory names that constitute the current search path. (Yours will look different, depending on your operating system, what version of Python you're running, and where it was originally installed.) Python will look through these directories (in this order) for a `.py` file matching the module name you're trying to import.

❸ Actually, I lied; the truth is more complicated than that, because not all modules are stored as `.py` files. Some, like the `sys` module, are "built−in modules"; they are actually baked right into Python itself. Built−in modules behave just like regular modules, but their Python source code is not available, because they are not written in Python! (The `sys` module is written in C.)

❹ You can add a new directory to Python's search path at runtime by appending the directory name to `sys.path`, and then Python will look in that directory as well, whenever you try to import a module. The effect lasts as long as Python is running. (You'll talk more about `append` and other list methods in Chapter 3.)

2.4.2. What's an Object?

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built−in attribute `__doc__`, which returns the `doc string` defined in the function's source code. The `sys` module is an object which has (among other things) an attribute called `path`. And so forth.

Still, this begs the question. What is an object? Different programming languages define "object" in different ways. In some, it means that *all* objects *must* have attributes and methods; in others, it means that all objects are subclassable. In Python, the definition is looser; some objects have neither attributes nor methods (more on this in Chapter 3), and not all objects are subclassable (more on this in Chapter 5). But everything is an object in the sense that it can be assigned to a variable or passed as an argument to a function (more in this in Chapter 4).

This is so important that I'm going to repeat it in case you missed it the first few times: *everything in Python is an object*. Strings are objects. Lists are objects. Functions are objects. Even modules are objects.

Further Reading on Objects

* *Python Reference Manual* (http://www.python.org/doc/current/ref/) explains exactly what it means to say that everything in Python is an object (http://www.python.org/doc/current/ref/objects.html), because some people are pedantic and like to discuss this sort of thing at great length.
* eff−bot (http://www.effbot.org/guides/) summarizes Python objects (http://www.effbot.org/guides/python−objects.htm).

2.5. Indenting Code

Python functions have no explicit `begin` or `end`, and no curly braces to mark where the function code starts and stops. The only delimiter is a colon (`:`) and the indentation of the code itself.

Example 2.5. Indenting the `buildConnectionString` Function

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

Code blocks are defined by their indentation. By "code block", I mean functions, `if` statements, `for` loops, `while` loops, and so forth. Indenting starts a block and unindenting ends it. There are no explicit braces, brackets, or keywords. This means that whitespace is significant, and must be consistent. In this example, the function code (including the `doc string`) is indented four spaces. It doesn't need to be four spaces, it just needs to be consistent. The first line that is not indented is outside the function.

Example 2.6, if Statements shows an example of code indentation with `if` statements.

Example 2.6. `if` Statements

```
def fib(n):                   ❶
    print 'n =', n            ❷
    if n > 1:                 ❸
        return n * fib(n − 1)
    else:                     ❹
        print 'end of the line'
        return 1
```

❶ This is a function named `fib` that takes one argument, `n`. All the code within the function is indented.

❷ Printing to the screen is very easy in Python, just use `print`. `print` statements can take any data type, including strings, integers, and other native types like dictionaries and lists that you'll learn about in the next chapter. You can even mix and match to print several things on one line by using a comma−separated list of values. Each value is printed on the same line, separated by spaces (the commas don't print). So when `fib` is called with `5`, this will print "n = 5".

❸ `if` statements are a type of code block. If the `if` expression evaluates to true, the indented block is executed, otherwise it falls to the `else` block.

❹ Of course `if` and `else` blocks can contain multiple lines, as long as they are all indented the same amount. This `else` block has two lines of code in it. There is no other special syntax for multi−line code blocks. Just indent and get on with your life.

After some initial protests and several snide analogies to Fortran, you will make peace with this and start seeing its benefits. One major benefit is that all Python programs look similar, since indentation is a language requirement and not a matter of style. This makes it easier to read and understand other people's Python code.

Python uses carriage returns to separate statements and a colon and indentation to separate code blocks. C++ and Java use semicolons to separate statements and curly braces to separate code blocks.

Further Reading on Code Indentation

- *Python Reference Manual* (http://www.python.org/doc/current/ref/) discusses cross−platform indentation issues and shows various indentation errors (http://www.python.org/doc/current/ref/indentation.html).
- *Python Style Guide* (http://www.python.org/doc/essays/styleguide.html) discusses good indentation style.

2.6. Testing Modules

Python modules are objects and have several useful attributes. You can use this to easily test your modules as you write them. Here's an example that uses the `if __name__` trick.

```
if __name__ == "__main__":
```

Some quick observations before you get to the good stuff. First, parentheses are not required around the `if` expression. Second, the `if` statement ends with a colon, and is followed by indented code.

Like C, Python uses `==` for comparison and `=` for assignment. Unlike C, Python does not support in−line assignment, so there's no chance of accidentally assigning the value you thought you were comparing.

So why is this particular `if` statement a trick? Modules are objects, and all modules have a built−in attribute `__name__`. A module's `__name__` depends on how you're using the module. If you `import` the module, then `__name__` is the module's filename, without a directory path or file extension. But you can also run the module directly as a standalone program, in which case `__name__` will be a special default value, `__main__`.

```
>>> import odbchelper
>>> odbchelper.__name__
'odbchelper'
```

Knowing this, you can design a test suite for your module within the module itself by putting it in this `if` statement. When you run the module directly, `__name__` is `__main__`, so the test suite executes. When you import the module, `__name__` is something else, so the test suite is ignored. This makes it easier to develop and debug new modules before integrating them into a larger program.

On MacPython, there is an additional step to make the `if __name__` trick work. Pop up the module's options menu by clicking the black triangle in the upper−right corner of the window, and make sure Run as __main__ is checked.

Further Reading on Importing Modules

- *Python Reference Manual* (http://www.python.org/doc/current/ref/) discusses the low−level details of importing modules (http://www.python.org/doc/current/ref/import.html).

Chapter 3. Native Datatypes

You'll get back to your first Python program in just a minute. But first, a short digression is in order, because you need to know about dictionaries, tuples, and lists (oh my!). If you're a Perl hacker, you can probably skim the bits about dictionaries and lists, but you should still pay attention to tuples.

3.1. Introducing Dictionaries

One of Python's built−in datatypes is the dictionary, which defines one−to−one relationships between keys and values.

A dictionary in Python is like a hash in Perl. In Perl, variables that store hashes always start with a `%` character. In Python, variables can be named anything, and Python keeps track of the datatype internally.

A dictionary in Python is like an instance of the `Hashtable` class in Java.

A dictionary in Python is like an instance of the `Scripting.Dictionary` object in Visual Basic.

3.1.1. Defining Dictionaries

Example 3.1. Defining a Dictionary

```
>>> d = {"server":"mpilgrim", "database":"master"} ❶
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["server"]                                    ❷
'mpilgrim'
>>> d["database"]                                  ❸
'master'
>>> d["mpilgrim"]                                  ❹
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
KeyError: mpilgrim
```

❶ First, you create a new dictionary with two elements and assign it to the variable `d`. Each element is a key−value pair, and the whole set of elements is enclosed in curly braces.

❷ `'server'` is a key, and its associated value, referenced by `d["server"]`, is `'mpilgrim'`.

❸ `'database'` is a key, and its associated value, referenced by `d["database"]`, is `'master'`.

❹ You can get values by key, but you can't get keys by value. So `d["server"]` is `'mpilgrim'`, but `d["mpilgrim"]` raises an exception, because `'mpilgrim'` is not a key.

3.1.2. Modifying Dictionaries

Example 3.2. Modifying a Dictionary

```
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["database"] = "pubs" ❶
>>> d
{'server': 'mpilgrim', 'database': 'pubs'}
>>> d["uid"] = "sa"        ❷
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
```

❶ You can not have duplicate keys in a dictionary. Assigning a value to an existing key will wipe out the old value.

❷ You can add new key−value pairs at any time. This syntax is identical to modifying existing values. (Yes, this will annoy you someday when you think you are adding new values but are actually just modifying the same value over and over because your key isn't changing the way you think it is.)

Note that the new element (key `'uid'`, value `'sa'`) appears to be in the middle. In fact, it was just a coincidence that the elements appeared to be in order in the first example; it is just as much a coincidence that they appear to be out of order now.

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered. This is an important distinction that will annoy you when you want to access the elements of a dictionary in a specific, repeatable order (like alphabetical order by key). There are ways of doing this, but they're not built into the dictionary.

When working with dictionaries, you need to be aware that dictionary keys are case−sensitive.

Example 3.3. Dictionary Keys Are Case−Sensitive

```
>>> d = {}
>>> d["key"] = "value"
>>> d["key"] = "other value" ❶
>>> d
{'key': 'other value'}
>>> d["Key"] = "third value" ❷
>>> d
{'Key': 'third value', 'key': 'other value'}
```

❶ Assigning a value to an existing dictionary key simply replaces the old value with a new one.

❷ This is not assigning a value to an existing dictionary key, because strings in Python are case−sensitive, so `'key'` is not the same as `'Key'`. This creates a new key/value pair in the dictionary; it may look similar to you, but as far as Python is concerned, it's completely different.

Example 3.4. Mixing Datatypes in a Dictionary

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
>>> d["retrycount"] = 3 ❶
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d[42] = "douglas"   ❷
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master',
42: 'douglas', 'retrycount': 3}
```

❶ Dictionaries aren't just for strings. Dictionary values can be any datatype, including strings, integers, objects, or even other dictionaries. And within a single dictionary, the values don't all need to be the same type; you can mix and match as needed.

❷ Dictionary keys are more restricted, but they can be strings, integers, and a few other types. You can also mix and match key datatypes within a dictionary.

3.1.3. Deleting Items From Dictionaries

Example 3.5. Deleting Items from a Dictionary

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master',
42: 'douglas', 'retrycount': 3}
>>> del d[42] ❶
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d.clear() ❷
>>> d
{}
```

❶ `del` lets you delete individual items from a dictionary by key.

❷ `clear` deletes all items from a dictionary. Note that the set of empty curly braces signifies a dictionary without any items.

Further Reading on Dictionaries

- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) teaches about dictionaries and shows how to use dictionaries to model sparse matrices (http://www.ibiblio.org/obp/thinkCSpy/chap10.htm).
- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) has a lot of example code using dictionaries (http://www.faqts.com/knowledge−base/index.phtml/fid/541).
- Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/) discusses how to sort the values of a dictionary by key (http://www.activestate.com/ASPN/Python/Cookbook/Recipe/52306).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes all the dictionary methods (http://www.python.org/doc/current/lib/typesmapping.html).

3.2. Introducing Lists

Lists are Python's workhorse datatype. If your only experience with lists is arrays in Visual Basic or (God forbid) the datastore in Powerbuilder, brace yourself for Python lists.

A list in Python is like an array in Perl. In Perl, variables that store arrays always start with the `@` character; in Python, variables can be named anything, and Python keeps track of the datatype internally.

A list in Python is much more than an array in Java (although it can be used as one if that's really all you want out of life). A better analogy would be to the `ArrayList` class, which can hold arbitrary objects and can expand dynamically as new items are added.

3.2.1. Defining Lists

Example 3.6. Defining a List

```
>>> li = ["a", "b", "mpilgrim", "z", "example"] ❶
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[0]                                       ❷
'a'
>>> li[4]                                       ❸
'example'
```

❶ First, you define a list of five elements. Note that they retain their original order. This is not an accident. A list is an ordered set of elements enclosed in square brackets.

❷ A list can be used like a zero−based array. The first element of any non−empty list is always `li[0]`.

❸ The last element of this five−element list is `li[4]`, because lists are always zero−based.

Example 3.7. Negative List Indices

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[-1] ❶
'example'
>>> li[-3] ❷
'mpilgrim'
```

❶ A negative index accesses elements from the end of the list counting backwards. The last element of any non−empty list is always `li[−1]`.

❷ If the negative index is confusing to you, think of it this way: `li[−n] == li[len(li) − n]`. So in this list, `li[−3] == li[5 − 3] == li[2]`.

Example 3.8. Slicing a List

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[1:3]  ❶
['b', 'mpilgrim']
>>> li[1:-1] ❷
['b', 'mpilgrim', 'z']
>>> li[0:3]  ❸
['a', 'b', 'mpilgrim']
```

❶ You can get a subset of a list, called a "slice", by specifying two indices. The return value is a new list containing all the elements of the list, in order, starting with the first slice index (in this case `li[1]`), up to but not including the second slice index (in this case `li[3]`).

❷ Slicing works if one or both of the slice indices is negative. If it helps, you can think of it this way: reading the list from left to right, the first slice index specifies the first element you want, and the second slice index specifies the first element you don't want. The return value is everything in between.

❸ Lists are zero−based, so `li[0:3]` returns the first three elements of the list, starting at `li[0]`, up to but not including `li[3]`.

Example 3.9. Slicing Shorthand

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[:3] ❶
['a', 'b', 'mpilgrim']
>>> li[3:] ❷ ❸
['z', 'example']
>>> li[:]  ❹
['a', 'b', 'mpilgrim', 'z', 'example']
```

❶ If the left slice index is 0, you can leave it out, and 0 is implied. So `li[:3]` is the same as `li[0:3]` from Example 3.8, Slicing a List.

❷ Similarly, if the right slice index is the length of the list, you can leave it out. So `li[3:]` is the same as `li[3:5]`, because this list has five elements.

❸ Note the symmetry here. In this five−element list, `li[:3]` returns the first 3 elements, and `li[3:]` returns the last two elements. In fact, `li[:n]` will always return the first `n` elements, and `li[n:]` will return the rest, regardless of the length of the list.

❹ If both slice indices are left out, all elements of the list are included. But this is not the same as the original `li` list; it is a new list that happens to have all the same elements. `li[:]` is shorthand for making a complete copy of a list.

3.2.2. Adding Elements to Lists

Example 3.10. Adding Elements to a List

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li.append("new")                  ❶
>>> li
['a', 'b', 'mpilgrim', 'z', 'example', 'new']
>>> li.insert(2, "new")               ❷
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new']
>>> li.extend(["two", "elements"]) ❸
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
```

❶ `append` adds a single element to the end of the list.

❷ `insert` inserts a single element into a list. The numeric argument is the index of the first element that gets bumped out of position. Note that list elements do not need to be unique; there are now two separate elements with the value `'new'`, `li[2]` and `li[6]`.

❸ `extend` concatenates lists. Note that you do not call `extend` with multiple arguments; you call it with one argument, a list. In this case, that list has two elements.

Example 3.11. The Difference between `extend` and `append`

```
>>> li = ['a', 'b', 'c']
>>> li.extend(['d', 'e', 'f']) ❶
>>> li
['a', 'b', 'c', 'd', 'e', 'f']
>>> len(li)                     ❷
6
>>> li[-1]
'f'
>>> li = ['a', 'b', 'c']
>>> li.append(['d', 'e', 'f']) ❸
>>> li
['a', 'b', 'c', ['d', 'e', 'f']]
>>> len(li)                     ❹
4
>>> li[-1]
['d', 'e', 'f']
```

❶ Lists have two methods, `extend` and `append`, that look like they do the same thing, but are in fact completely different. `extend` takes a single argument, which is always a list, and adds each of the elements of that list to the original list.

❷ Here you started with a list of three elements (`'a'`, `'b'`, and `'c'`), and you extended the list with a list of another three elements (`'d'`, `'e'`, and `'f'`), so you now have a list of six elements.

❸

On the other hand, `append` takes one argument, which can be any data type, and simply adds it to the end of the list. Here, you're calling the `append` method with a single argument, which is a list of three elements.

❹ Now the original list, which started as a list of three elements, contains four elements. Why four? Because the last element that you just appended *is itself a list*. Lists can contain any type of data, including other lists. That may be what you want, or maybe not. Don't use `append` if you mean `extend`.

3.2.3. Searching Lists

Example 3.12. Searching a List

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.index("example") ❶
5
>>> li.index("new")     ❷
2
>>> li.index("c")       ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in li           ❹
False
```

❶ `index` finds the first occurrence of a value in the list and returns the index.

❷ `index` finds the *first* occurrence of a value in the list. In this case, `'new'` occurs twice in the list, in `li[2]` and `li[6]`, but `index` will return only the first index, `2`.

❸ If the value is not found in the list, Python raises an exception. This is notably different from most languages, which will return some invalid index. While this may seem annoying, it is a good thing, because it means your program will crash at the source of the problem, rather than later on when you try to use the invalid index.

❹ To test whether a value is in the list, use `in`, which returns `True` if the value is found or `False` if it is not.

Before version 2.2.1, Python had no separate boolean datatype. To compensate for this, Python accepted almost anything in a boolean context (like an `if` statement), according to the following rules:

- `0` is false; all other numbers are true.
- An empty string (`""`) is false, all other strings are true.
- An empty list (`[]`) is false; all other lists are true.
- An empty tuple (`()`) is false; all other tuples are true.
- An empty dictionary (`{}`) is false; all other dictionaries are true.

These rules still apply in Python 2.2.1 and beyond, but now you can also use an actual boolean, which has a value of `True` or `False`. Note the capitalization; these values, like everything else in Python, are case−sensitive.

3.2.4. Deleting List Elements

Example 3.13. Removing Elements from a List

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.remove("z")   ❶
>>> li
['a', 'b', 'new', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("new") ❷
```

```
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("c")   ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> li.pop()         ❹
'elements'
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
```

❶ `remove` removes the first occurrence of a value from a list.

❷ `remove` removes *only* the first occurrence of a value. In this case, `'new'` appeared twice in the list, but `li.remove("new")` removed only the first occurrence.

❸ If the value is not found in the list, Python raises an exception. This mirrors the behavior of the `index` method.

❹ `pop` is an interesting beast. It does two things: it removes the last element of the list, and it returns the value that it removed. Note that this is different from `li[−1]`, which returns a value but does not change the list, and different from `li.remove(*value*)`, which changes the list but does not return a value.

3.2.5. Using List Operators

Example 3.14. List Operators

```
>>> li = ['a', 'b', 'mpilgrim']
>>> li = li + ['example', 'new'] ❶
>>> li
['a', 'b', 'mpilgrim', 'example', 'new']
>>> li += ['two']                ❷
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
>>> li = [1, 2] * 3              ❸
>>> li
[1, 2, 1, 2, 1, 2]
```

❶ Lists can also be concatenated with the `+` operator. `*list* = *list* + *otherlist*` has the same result as `*list*.extend(*otherlist*)`. But the `+` operator returns a new (concatenated) list as a value, whereas `extend` only alters an existing list. This means that `extend` is faster, especially for large lists.

❷ Python supports the `+=` operator. `li += ['two']` is equivalent to `li.extend(['two'])`. The `+=` operator works for lists, strings, and integers, and it can be overloaded to work for user−defined classes as well. (More on classes in Chapter 5.)

❸ The `*` operator works on lists as a repeater. `li = [1, 2] * 3` is equivalent to `li = [1, 2] + [1, 2] + [1, 2]`, which concatenates the three lists into one.

Further Reading on Lists

- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) teaches about lists and makes an important point about passing lists as function arguments (http://www.ibiblio.org/obp/thinkCSpy/chap08.htm).
- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) shows how to use lists as stacks and queues (http://www.python.org/doc/current/tut/node7.html#SECTION007110000000000000000).
- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) answers common questions about lists (http://www.faqts.com/knowledge−base/index.phtml/fid/534) and has a lot of example code using lists (http://www.faqts.com/knowledge−base/index.phtml/fid/540).

* *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes all the list methods (http://www.python.org/doc/current/lib/typesseq−mutable.html).

3.3. Introducing Tuples

A tuple is an immutable list. A tuple can not be changed in any way once it is created.

Example 3.15. Defining a tuple

```
>>> t = ("a", "b", "mpilgrim", "z", "example") ❶
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t[0]                                       ❷
'a'
>>> t[-1]                                      ❸
'example'
>>> t[1:3]                                     ❹
('b', 'mpilgrim')
```

❶ A tuple is defined in the same way as a list, except that the whole set of elements is enclosed in parentheses instead of square brackets.

❷ The elements of a tuple have a defined order, just like a list. Tuples indices are zero−based, just like a list, so the first element of a non−empty tuple is always `t[0]`.

❸ Negative indices count from the end of the tuple, just as with a list.

❹ Slicing works too, just like a list. Note that when you slice a list, you get a new list; when you slice a tuple, you get a new tuple.

Example 3.16. Tuples Have No Methods

```
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t.append("new")    ❶
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove("z")      ❷
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> t.index("example") ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'index'
>>> "z" in t           ❹
True
```

❶ You can't add elements to a tuple. Tuples have no `append` or `extend` method.

❷ You can't remove elements from a tuple. Tuples have no `remove` or `pop` method.

❸ You can't find elements in a tuple. Tuples have no `index` method.

❹ You can, however, use `in` to see if an element exists in the tuple.

So what are tuples good for?

* Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is

iterate through it, use a tuple instead of a list.
- It makes your code safer if you "write−protect" data that does not need to be changed. Using a tuple instead of a list is like having an implied `assert` statement that shows this data is constant, and that special thought (and a specific function) is required to override that.
- Remember that I said that dictionary keys can be integers, strings, and "a few other types"? Tuples are one of those types. Tuples can be used as keys in a dictionary, but lists can't be used this way.Actually, it's more complicated than that. Dictionary keys must be immutable. Tuples themselves are immutable, but if you have a tuple of lists, that counts as mutable and isn't safe to use as a dictionary key. Only tuples of strings, numbers, or other dictionary−safe tuples can be used as dictionary keys.
- Tuples are used in string formatting, as you'll see shortly.

Tuples can be converted into lists, and vice−versa. The built−in `tuple` function takes a list and returns a tuple with the same elements, and the `list` function takes a tuple and returns a list. In effect, `tuple` freezes a list, and `list` thaws a tuple.

Further Reading on Tuples

- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) teaches about tuples and shows how to concatenate tuples (http://www.ibiblio.org/obp/thinkCSpy/chap10.htm).
- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) shows how to sort a tuple (http://www.faqts.com/knowledge−base/view.phtml/aid/4553/fid/587).
- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) shows how to define a tuple with one element (http://www.python.org/doc/current/tut/node7.html#SECTION007300000000000000000).

3.4. Declaring variables

Now that you know something about dictionaries, tuples, and lists (oh my!), let's get back to the sample program from Chapter 2, `odbchelper.py`.

Python has local and global variables like most other languages, but it has no explicit variable declarations. Variables spring into existence by being assigned a value, and they are automatically destroyed when they go out of scope.

Example 3.17. Defining the `myParams` Variable

```
if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
```

Notice the indentation. An `if` statement is a code block and needs to be indented just like a function.

Also notice that the variable assignment is one command split over several lines, with a backslash ("\") serving as a line−continuation marker.

When a command is split among several lines with the line−continuation marker ("\"), the continued lines can be indented in any manner; Python's normally stringent indentation rules do not apply. If your Python IDE auto−indents the continued line, you should probably accept its default unless you have a burning reason not to.

Strictly speaking, expressions in parentheses, straight brackets, or curly braces (like defining a dictionary) can be split into multiple lines with or without the line continuation character ("\"). I like to include the backslash even when it's not required because I think it makes the code easier to read, but that's a matter of style.

Third, you never declared the variable `myParams`, you just assigned a value to it. This is like VBScript without the `option explicit` option. Luckily, unlike VBScript, Python will not allow you to reference a variable that has never been assigned a value; trying to do so will raise an exception.

3.4.1. Referencing Variables

Example 3.18. Referencing an Unbound Variable

```
>>> x
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'x'
>>> x = 1
>>> x
1
```

You will thank Python for this one day.

3.4.2. Assigning Multiple Values at Once

One of the cooler programming shortcuts in Python is using sequences to assign multiple values at once.

Example 3.19. Assigning multiple values at once

```
>>> v = ('a', 'b', 'e')
>>> (x, y, z) = v      ❶
>>> x
'a'
>>> y
'b'
>>> z
'e'
```

❶ `v` is a tuple of three elements, and `(x, y, z)` is a tuple of three variables. Assigning one to the other assigns each of the values of `v` to each of the variables, in order.

This has all sorts of uses. I often want to assign names to a range of values. In C, you would use `enum` and manually list each constant and its associated value, which seems especially tedious when the values are consecutive. In Python, you can use the built−in `range` function with multi−variable assignment to quickly assign consecutive values.

Example 3.20. Assigning Consecutive Values

```
>>> range(7)                                                                  ❶
[0, 1, 2, 3, 4, 5, 6]
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7) ❷
>>> MONDAY                                                                    ❸
0
>>> TUESDAY
1
>>> SUNDAY
```

```
6
```

1. The built−in `range` function returns a list of integers. In its simplest form, it takes an upper limit and returns a zero−based list counting up to but not including the upper limit. (If you like, you can pass other parameters to specify a base other than `0` and a step other than `1`. You can `print range.__doc__` for details.)
2. `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, and `SUNDAY` are the variables you're defining. (This example came from the `calendar` module, a fun little module that prints calendars, like the UNIX program `cal`. The `calendar` module defines integer constants for days of the week.)
3. Now each variable has its value: `MONDAY` is `0`, `TUESDAY` is `1`, and so forth.

You can also use multi−variable assignment to build functions that return multiple values, simply by returning a tuple of all the values. The caller can treat it as a tuple, or assign the values to individual variables. Many standard Python libraries do this, including the `os` module, which you'll discuss in Chapter 6.

Further Reading on Variables

- *Python Reference Manual* (http://www.python.org/doc/current/ref/) shows examples of when you can skip the line continuation character (http://www.python.org/doc/current/ref/implicit−joining.html) and when you need to use it (http://www.python.org/doc/current/ref/explicit−joining.html).
- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) shows how to use multi−variable assignment to swap the values of two variables (http://www.ibiblio.org/obp/thinkCSpy/chap09.htm).

3.5. Formatting Strings

Python supports formatting values into strings. Although this can include very complicated expressions, the most basic usage is to insert values into a string with the `%s` placeholder.

String formatting in Python uses the same syntax as the `sprintf` function in C.

Example 3.21. Introducing String Formatting

```
>>> k = "uid"
>>> v = "sa"
>>> "%s=%s" % (k, v) ❶
'uid=sa'
```

1. The whole expression evaluates to a string. The first `%s` is replaced by the value of `k`; the second `%s` is replaced by the value of `v`. All other characters in the string (in this case, the equal sign) stay as they are.

Note that `(k, v)` is a tuple. I told you they were good for something.

You might be thinking that this is a lot of work just to do simple string concatentation, and you would be right, except that string formatting isn't just concatenation. It's not even just formatting. It's also type coercion.

Example 3.22. String Formatting vs. Concatenating

```
>>> uid = "sa"
>>> pwd = "secret"
>>> print pwd + " is not a good password for " + uid      ❶
secret is not a good password for sa
>>> print "%s is not a good password for %s" % (pwd, uid) ❷
```

```
secret is not a good password for sa
>>> userCount = 6
>>> print "Users connected: %d" % (userCount, )          ❸ ❹
Users connected: 6
>>> print "Users connected: " + userCount                ❺
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

❶ `+` is the string concatenation operator.

❷ In this trivial case, string formatting accomplishes the same result as concatentation.

❸ `(userCount, )` is a tuple with one element. Yes, the syntax is a little strange, but there's a good reason for it: it's unambiguously a tuple. In fact, you can always include a comma after the last element when defining a list, tuple, or dictionary, but the comma is required when defining a tuple with one element. If the comma weren't required, Python wouldn't know whether `(userCount)` was a tuple with one element or just the value of `userCount`.

❹ String formatting works with integers by specifying `%d` instead of `%s`.

❺ Trying to concatenate a string with a non−string raises an exception. Unlike string formatting, string concatenation works only when everything is already a string.

As with `printf` in C, string formatting in Python is like a Swiss Army knife. There are options galore, and modifier strings to specially format many different types of values.

Example 3.23. Formatting Numbers

```
>>> print "Today's stock price: %f" % 50.4625   ❶
50.462500
>>> print "Today's stock price: %.2f" % 50.4625 ❷
50.46
>>> print "Change since yesterday: %+.2f" % 1.5 ❸
+1.50
```

❶ The `%f` string formatting option treats the value as a decimal, and prints it to six decimal places.

❷ The ".2" modifier of the `%f` option truncates the value to two decimal places.

❸ You can even combine modifiers. Adding the `+` modifier displays a plus or minus sign before the value. Note that the ".2" modifier is still in place, and is padding the value to exactly two decimal places.

Further Reading on String Formatting

- *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes all the string formatting format characters (http://www.python.org/doc/current/lib/typesseq−strings.html).
- *Effective AWK Programming* (http://www−gnats.gnu.org:8080/cgi−bin/info2www?(gawk)Top) discusses all the format characters (http://www−gnats.gnu.org:8080/cgi−bin/info2www?(gawk)Control+Letters) and advanced string formatting techniques like specifying width, precision, and zero−padding (http://www−gnats.gnu.org:8080/cgi−bin/info2www?(gawk)Format+Modifiers).

3.6. Mapping Lists

One of the most powerful features of Python is the list comprehension, which provides a compact way of mapping a list into another list by applying a function to each of the elements of the list.

Example 3.24. Introducing List Comprehensions

```
>>> li = [1, 9, 8, 4]
>>> [elem*2 for elem in li]       ❶
[2, 18, 16, 8]
>>> li                            ❷
[1, 9, 8, 4]
>>> li = [elem*2 for elem in li] ❸
>>> li
[2, 18, 16, 8]
```

❶ To make sense of this, look at it from right to left. `li` is the list you're mapping. Python loops through `li` one element at a time, temporarily assigning the value of each element to the variable `elem`. Python then applies the function `elem*2` and appends that result to the returned list.

❷ Note that list comprehensions do not change the original list.

❸ It is safe to assign the result of a list comprehension to the variable that you're mapping. Python constructs the new list in memory, and when the list comprehension is complete, it assigns the result to the variable.

Here are the list comprehensions in the `buildConnectionString` function that you declared in Chapter 2:

```
["%s=%s" % (k, v) for k, v in params.items()]
```

First, notice that you're calling the `items` function of the `params` dictionary. This function returns a list of tuples of all the data in the dictionary.

Example 3.25. The `keys`, `values`, and `items` Functions

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.keys()   ❶
['server', 'uid', 'database', 'pwd']
>>> params.values() ❷
['mpilgrim', 'sa', 'master', 'secret']
>>> params.items()  ❸
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd', 'secret')]
```

❶ The `keys` method of a dictionary returns a list of all the keys. The list is not in the order in which the dictionary was defined (remember that elements in a dictionary are unordered), but it is a list.

❷ The `values` method returns a list of all the values. The list is in the same order as the list returned by `keys`, so `params.values()[n] == params[params.keys()[n]]` for all values of `n`.

❸ The `items` method returns a list of tuples of the form `(`*`key`*`, `*`value`*`)`. The list contains all the data in the dictionary.

Now let's see what `buildConnectionString` does. It takes a list, `params.items()`, and maps it to a new list by applying string formatting to each element. The new list will have the same number of elements as `params.items()`, but each element in the new list will be a string that contains both a key and its associated value from the `params` dictionary.

Example 3.26. List Comprehensions in `buildConnectionString`, Step by Step

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.items()
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd', 'secret')]
>>> [k for k, v in params.items()]                ❶
['server', 'uid', 'database', 'pwd']
>>> [v for k, v in params.items()]                ❷
```

```
['mpilgrim', 'sa', 'master', 'secret']
>>> ["%s=%s" % (k, v) for k, v in params.items()] ❸
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
```

❶ Note that you're using two variables to iterate through the `params.items()` list. This is another use of multi−variable assignment. The first element of `params.items()` is `('server', 'mpilgrim')`, so in the first iteration of the list comprehension, `k` will get `'server'` and `v` will get `'mpilgrim'`. In this case, you're ignoring the value of `v` and only including the value of `k` in the returned list, so this list comprehension ends up being equivalent to `params.keys()`.

❷ Here you're doing the same thing, but ignoring the value of `k`, so this list comprehension ends up being equivalent to `params.values()`.

❸ Combining the previous two examples with some simple string formatting, you get a list of strings that include both the key and value of each element of the dictionary. This looks suspiciously like the output of the program. All that remains is to join the elements in this list into a single string.

Further Reading on List Comprehensions

- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses another way to map lists using the built−in `map` function (http://www.python.org/doc/current/tut/node7.html#SECTION007130000000000000000).
- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) shows how to do nested list comprehensions (http://www.python.org/doc/current/tut/node7.html#SECTION007140000000000000000).

3.7. Joining Lists and Splitting Strings

You have a list of key−value pairs in the form *`key=value`*, and you want to join them into a single string. To join any list of strings into a single string, use the `join` method of a string object.

Here is an example of joining a list from the `buildConnectionString` function:

```
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

One interesting note before you continue. I keep repeating that functions are objects, strings are objects... everything is an object. You might have thought I meant that string *variables* are objects. But no, look closely at this example and you'll see that the string `";"` itself is an object, and you are calling its `join` method.

The `join` method joins the elements of the list into a single string, with each element separated by a semi−colon. The delimiter doesn't need to be a semi−colon; it doesn't even need to be a single character. It can be any string.

`join` works only on lists of strings; it does not do any type coercion. Joining a list that has one or more non−string elements will raise an exception.

Example 3.27. Output of `odbchelper.py`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> ["%s=%s" % (k, v) for k, v in params.items()]
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> ";".join(["%s=%s" % (k, v) for k, v in params.items()])
'server=mpilgrim;uid=sa;database=master;pwd=secret'
```

This string is then returned from the `odbchelper` function and printed by the calling block, which gives you the output that you marveled at when you started reading this chapter.

You're probably wondering if there's an analogous method to split a string into a list. And of course there is, and it's called `split`.

Example 3.28. Splitting a String

```
>>> li = ['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s = ";".join(li)
>>> s
'server=mpilgrim;uid=sa;database=master;pwd=secret'
>>> s.split(";")    ❶
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s.split(";", 1) ❷
['server=mpilgrim', 'uid=sa;database=master;pwd=secret']
```

❶ `split` reverses `join` by splitting a string into a multi−element list. Note that the delimiter (";") is stripped out completely; it does not appear in any of the elements of the returned list.

❷ `split` takes an optional second argument, which is the number of times to split. (""Oooooh, optional arguments..." You'll learn how to do this in your own functions in the next chapter.)

`*anystring*.split(*delimiter*, 1)` is a useful technique when you want to search a string for a substring and then work with everything before the substring (which ends up in the first element of the returned list) and everything after it (which ends up in the second element).

Further Reading on String Methods

- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) answers common questions about strings (http://www.faqts.com/knowledge−base/index.phtml/fid/480) and has a lot of example code using strings (http://www.faqts.com/knowledge−base/index.phtml/fid/539).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes all the string methods (http://www.python.org/doc/current/lib/string−methods.html).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the `string` module (http://www.python.org/doc/current/lib/module−string.html).
- *The Whole Python FAQ* (http://www.python.org/doc/FAQ.html) explains why `join` is a string method (http://www.python.org/cgi−bin/faqw.py?query=4.96&querytype=simple&casefold=yes&req=search) instead of a list method.

3.7.1. Historical Note on String Methods

When I first learned Python, I expected `join` to be a method of a list, which would take the delimiter as an argument. Many people feel the same way, and there's a story behind the `join` method. Prior to Python 1.6, strings didn't have all these useful methods. There was a separate `string` module that contained all the string functions; each function took a string as its first argument. The functions were deemed important enough to put onto the strings themselves, which made sense for functions like `lower`, `upper`, and `split`. But many hard−core Python programmers objected to the new `join` method, arguing that it should be a method of the list instead, or that it shouldn't move at all but simply stay a part of the old `string` module (which still has a lot of useful stuff in it). I use the new `join` method exclusively, but you will see code written either way, and if it really bothers you, you can use the old `string.join` function instead.

3.8. Summary

The `odbchelper.py` program and its output should now make perfect sense.

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.
```

```
    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

Here is the output of `odbchelper.py`:

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

Before diving into the next chapter, make sure you're comfortable doing all of these things:

- Using the Python IDE to test expressions interactively
- Writing Python programs and running them from within your IDE, or from the command line
- Importing modules and calling their functions
- Declaring functions and using `doc strings`, local variables, and proper indentation
- Defining dictionaries, tuples, and lists
- Accessing attributes and methods of any object, including strings, lists, dictionaries, functions, and modules
- Concatenating values through string formatting
- Mapping lists into other lists using list comprehensions
- Splitting strings into lists and joining lists into strings

Chapter 4. The Power Of Introspection

This chapter covers one of Python's strengths: introspection. As you know, everything in Python is an object, and introspection is code looking at other modules and functions in memory as objects, getting information about them, and manipulating them. Along the way, you'll define functions with no name, call functions with arguments out of order, and reference functions whose names you don't even know ahead of time.

4.1. Diving In

Here is a complete, working Python program. You should understand a good deal about it just by looking at it. The numbered lines illustrate concepts covered in Chapter 2, *Your First Python Program*. Don't worry if the rest of the code looks intimidating; you'll learn all about it throughout this chapter.

Example 4.1. `apihelper.py`

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
def info(object, spacing=10, collapse=1): ❶ ❷ ❸
    """Print methods and doc strings.

    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                      (method.ljust(spacing),
                       processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])

if __name__ == "__main__":                ❹ ❺
    print info.__doc__
```

❶ This module has one function, `info`. According to its function declaration, it takes three parameters: `object`, `spacing`, and `collapse`. The last two are actually optional parameters, as you'll see shortly.

❷ The `info` function has a multi−line `doc string` that succinctly describes the function's purpose. Note that no return value is mentioned; this function will be used solely for its effects, rather than its value.

❸ Code within the function is indented.

❹ The `if __name__` trick allows this program do something useful when run by itself, without interfering with its use as a module for other programs. In this case, the program simply prints out the `doc string` of the `info` function.

❺ `if` statements use `==` for comparison, and parentheses are not required.

The `info` function is designed to be used by you, the programmer, while working in the Python IDE. It takes any object that has functions or methods (like a module, which has functions, or a list, which has methods) and prints out the functions and their `doc string`s.

Example 4.2. Sample Usage of `apihelper.py`

```
>>> from apihelper import info
>>> li = []
>>> info(li)
append     L.append(object) -- append object to end
```

```
count      L.count(value) -> integer -- return number of occurrences of value
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1, 0, 1
```

By default the output is formatted to be easy to read. Multi−line `doc string`s are collapsed into a single long line, but this option can be changed by specifying `0` for the *`collapse`* argument. If the function names are longer than 10 characters, you can specify a larger value for the *`spacing`* argument to make the output easier to read.

Example 4.3. Advanced Usage of `apihelper.py`

```
>>> import odbchelper
>>> info(odbchelper)
buildConnectionString Build a connection string from a dictionary Returns string.
>>> info(odbchelper, 30)
buildConnectionString          Build a connection string from a dictionary Returns string.
>>> info(odbchelper, 30, 0)
buildConnectionString          Build a connection string from a dictionary

    Returns string.
```

4.2. Using Optional and Named Arguments

Python allows function arguments to have default values; if the function is called without the argument, the argument gets its default value. Futhermore, arguments can be specified in any order by using named arguments. Stored procedures in SQL Server Transact/SQL can do this, so if you're a SQL Server scripting guru, you can skim this part.

Here is an example of `info`, a function with two optional arguments:

```
def info(object, spacing=10, collapse=1):
```

`spacing` and `collapse` are optional, because they have default values defined. `object` is required, because it has no default value. If `info` is called with only one argument, `spacing` defaults to `10` and `collapse` defaults to `1`. If `info` is called with two arguments, `collapse` still defaults to `1`.

Say you want to specify a value for `collapse` but want to accept the default value for `spacing`. In most languages, you would be out of luck, because you would need to call the function with three arguments. But in Python, arguments can be specified by name, in any order.

Example 4.4. Valid Calls of `info`

```
info(odbchelper)                    ❶
info(odbchelper, 12)                ❷
info(odbchelper, collapse=0)        ❸
info(spacing=15, object=odbchelper) ❹
```

❶ With only one argument, `spacing` gets its default value of `10` and `collapse` gets its default value of `1`.

❷ With two arguments, `collapse` gets its default value of `1`.

❸ Here you are naming the `collapse` argument explicitly and specifying its value. `spacing` still gets its default value of `10`.

❹ Even required arguments (like `object`, which has no default value) can be named, and named arguments can appear in any order.

This looks totally whacked until you realize that arguments are simply a dictionary. The "normal" method of calling functions without argument names is actually just a shorthand where Python matches up the values with the argument names in the order they're specified in the function declaration. And most of the time, you'll call functions the "normal" way, but you always have the additional flexibility if you need it.

The only thing you need to do to call a function is specify a value (somehow) for each required argument; the manner and order in which you do that is up to you.

Further Reading on Optional Arguments

- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses exactly when and how default arguments are evaluated (http://www.python.org/doc/current/tut/node6.html#SECTION006710000000000000000), which matters when the default value is a list or an expression with side effects.

4.3. Using `type`, `str`, `dir`, and Other Built−In Functions

Python has a small set of extremely useful built−in functions. All other functions are partitioned off into modules. This was actually a conscious design decision, to keep the core language from getting bloated like other scripting languages (cough cough, Visual Basic).

4.3.1. The `type` Function

The `type` function returns the datatype of any arbitrary object. The possible types are listed in the `types` module. This is useful for helper functions that can handle several types of data.

Example 4.5. Introducing `type`

```
>>> type(1)           ❶
<type 'int'>
>>> li = []
>>> type(li)          ❷
<type 'list'>
>>> import odbchelper
>>> type(odbchelper)  ❸
<type 'module'>
>>> import types      ❹
>>> type(odbchelper) == types.ModuleType
True
```

❶ `type` takes anything −− and I mean anything −− and returns its datatype. Integers, strings, lists, dictionaries, tuples, functions, classes, modules, even types are acceptable.

❷ `type` can take a variable and return its datatype.

❸ `type` also works on modules.

❹ You can use the constants in the `types` module to compare types of objects. This is what the `info` function does, as you'll see shortly.

4.3.2. The `str` Function

The `str` coerces data into a string. Every datatype can be coerced into a string.

Example 4.6. Introducing `str`

```
>>> str(1)          ❶
'1'
>>> horsemen = ['war', 'pestilence', 'famine']
>>> horsemen
['war', 'pestilence', 'famine']
>>> horsemen.append('Powerbuilder')
>>> str(horsemen)   ❷
"['war', 'pestilence', 'famine', 'Powerbuilder']"
>>> str(odbchelper) ❸
"<module 'odbchelper' from 'c:\\docbook\\dip\\py\\odbchelper.py'>"
>>> str(None)       ❹
'None'
```

❶ For simple datatypes like integers, you would expect `str` to work, because almost every language has a function to convert an integer to a string.

❷ However, `str` works on any object of any type. Here it works on a list which you've constructed in bits and pieces.

❸ `str` also works on modules. Note that the string representation of the module includes the pathname of the module on disk, so yours will be different.

❹ A subtle but important behavior of `str` is that it works on `None`, the Python null value. It returns the string `'None'`. You'll use this to your advantage in the `info` function, as you'll see shortly.

At the heart of the `info` function is the powerful `dir` function. `dir` returns a list of the attributes and methods of any object: modules, functions, strings, lists, dictionaries... pretty much anything.

Example 4.7. Introducing `dir`

```
>>> li = []
>>> dir(li)           ❶
['append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
>>> d = {}
>>> dir(d)            ❷
['clear', 'copy', 'get', 'has_key', 'items', 'keys', 'setdefault', 'update', 'values']
>>> import odbchelper
>>> dir(odbchelper)   ❸
['__builtins__', '__doc__', '__file__', '__name__', 'buildConnectionString']
```

❶ `li` is a list, so `dir(li)` returns a list of all the methods of a list. Note that the returned list contains the names of the methods as strings, not the methods themselves.

❷ `d` is a dictionary, so `dir(d)` returns a list of the names of dictionary methods. At least one of these, `keys`, should look familiar.

❸ This is where it really gets interesting. `odbchelper` is a module, so `dir(odbchelper)` returns a list of all kinds of stuff defined in the module, including built−in attributes, like `__name__`, `__doc__`, and whatever other attributes and methods you define. In this case, `odbchelper` has only one user−defined method, the `buildConnectionString` function described in Chapter 2.

Finally, the `callable` function takes any object and returns `True` if the object can be called, or `False` otherwise.

Callable objects include functions, class methods, even classes themselves. (More on classes in the next chapter.)

Example 4.8. Introducing `callable`

```
>>> import string
>>> string.punctuation           [1]
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.join                  [2]
<function join at 00C55A7C>
>>> callable(string.punctuation) [3]
False
>>> callable(string.join)        [4]
True
>>> print string.join.__doc__    [5]
join(list [,sep]) -> string

    Return a string composed of the words in list, with
    intervening occurrences of sep.  The default separator is a
    single space.

    (joinfields and join are synonymous)
```

[1] The functions in the `string` module are deprecated (although many people still use the `join` function), but the module contains a lot of useful constants like this `string.punctuation`, which contains all the standard punctuation characters.

[2] `string.join` is a function that joins a list of strings.

[3] `string.punctuation` is not callable; it is a string. (A string does have callable methods, but the string itself is not callable.)

[4] `string.join` is callable; it's a function that takes two arguments.

[5] Any callable object may have a `doc string`. By using the `callable` function on each of an object's attributes, you can determine which attributes you care about (methods, functions, classes) and which you want to ignore (constants and so on) without knowing anything about the object ahead of time.

4.3.3. Built−In Functions

`type`, `str`, `dir`, and all the rest of Python's built−in functions are grouped into a special module called `__builtin__`. (That's two underscores before and after.) If it helps, you can think of Python automatically executing `from __builtin__ import *` on startup, which imports all the "built−in" functions into the namespace so you can use them directly.

The advantage of thinking like this is that you can access all the built−in functions and attributes as a group by getting information about the `__builtin__` module. And guess what, Python has a function called `info`. Try it yourself and skim through the list now. We'll dive into some of the more important functions later. (Some of the built−in error classes, like `AttributeError`, should already look familiar.)

Example 4.9. Built−in Attributes and Functions

```
>>> from apihelper import info
>>> import __builtin__
>>> info(__builtin__, 20)
ArithmeticError      Base class for arithmetic errors.
AssertionError       Assertion failed.
AttributeError       Attribute not found.
```

```
EOFError              Read beyond end of file.
EnvironmentError      Base class for I/O related errors.
Exception             Common base class for all exceptions.
FloatingPointError    Floating point operation failed.
IOError               I/O operation failed.

[...snip...]
```

Python comes with excellent reference manuals, which you should peruse thoroughly to learn all the modules Python has to offer. But unlike most languages, where you would find yourself referring back to the manuals or man pages to remind yourself how to use these modules, Python is largely self−documenting.

Further Reading on Built−In Functions

- *Python Library Reference* (http://www.python.org/doc/current/lib/) documents all the built−in functions (http://www.python.org/doc/current/lib/built−in−funcs.html) and all the built−in exceptions (http://www.python.org/doc/current/lib/module−exceptions.html).

4.4. Getting Object References With `getattr`

You already know that Python functions are objects. What you don't know is that you can get a reference to a function without knowing its name until run−time, by using the `getattr` function.

Example 4.10. Introducing `getattr`

```
>>> li = ["Larry", "Curly"]
>>> li.pop                       (1)
<built-in method pop of list object at 010DF884>
>>> getattr(li, "pop")           (2)
<built-in method pop of list object at 010DF884>
>>> getattr(li, "append")("Moe") (3)
>>> li
["Larry", "Curly", "Moe"]
>>> getattr({}, "clear")         (4)
<built-in method clear of dictionary object at 00F113D4>
>>> getattr((), "pop")           (5)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'pop'
```

(1) This gets a reference to the `pop` method of the list. Note that this is not calling the `pop` method; that would be `li.pop()`. This is the method itself.

(2) This also returns a reference to the `pop` method, but this time, the method name is specified as a string argument to the `getattr` function. `getattr` is an incredibly useful built−in function that returns any attribute of any object. In this case, the object is a list, and the attribute is the `pop` method.

(3) In case it hasn't sunk in just how incredibly useful this is, try this: the return value of `getattr` *is* the method, which you can then call just as if you had said `li.append("Moe")` directly. But you didn't call the function directly; you specified the function name as a string instead.

(4) `getattr` also works on dictionaries.

(5) In theory, `getattr` would work on tuples, except that tuples have no methods, so `getattr` will raise an exception no matter what attribute name you give.

4.4.1. `getattr` with Modules

`getattr` isn't just for built−in datatypes. It also works on modules.

Example 4.11. The `getattr` Function in `apihelper.py`

```
>>> import odbchelper
>>> odbchelper.buildConnectionString             ❶
<function buildConnectionString at 00D18DD4>
>>> getattr(odbchelper, "buildConnectionString") ❷
<function buildConnectionString at 00D18DD4>
>>> object = odbchelper
>>> method = "buildConnectionString"
>>> getattr(object, method)                      ❸
<function buildConnectionString at 00D18DD4>
>>> type(getattr(object, method))                ❹
<type 'function'>
>>> import types
>>> type(getattr(object, method)) == types.FunctionType
True
>>> callable(getattr(object, method))            ❺
True
```

❶ This returns a reference to the `buildConnectionString` function in the `odbchelper` module, which you studied in Chapter 2, *Your First Python Program*. (The hex address you see is specific to my machine; your output will be different.)

❷ Using `getattr`, you can get the same reference to the same function. In general, `getattr(`*`object`*`, "`*`attribute`*`")` is equivalent to *`object.attribute`*. If *`object`* is a module, then *`attribute`* can be anything defined in the module: a function, class, or global variable.

❸ And this is what you actually use in the `info` function. `object` is passed into the function as an argument; `method` is a string which is the name of a method or function.

❹ In this case, `method` is the name of a function, which you can prove by getting its `type`.

❺ Since `method` is a function, it is callable.

4.4.2. `getattr` As a Dispatcher

A common usage pattern of `getattr` is as a dispatcher. For example, if you had a program that could output data in a variety of different formats, you could define separate functions for each output format and use a single dispatch function to call the right one.

For example, let's imagine a program that prints site statistics in HTML, XML, and plain text formats. The choice of output format could be specified on the command line, or stored in a configuration file. A `statsout` module defines three functions, `output_html`, `output_xml`, and `output_text`. Then the main program defines a single output function, like this:

Example 4.12. Creating a Dispatcher with `getattr`

```
import statsout

def output(data, format="text"):                              ❶
    output_function = getattr(statsout, "output_%s" % format) ❷
    return output_function(data)                              ❸
```

❶ The `output` function takes one required argument, `data`, and one optional argument, `format`. If `format` is not specified, it defaults to `text`, and you will end up calling the plain text output function.

❷ You concatenate the `format` argument with "output_" to produce a function name, and then go get that function from the `statsout` module. This allows you to easily extend the program later to support other output formats, without changing this dispatch function. Just add another function to `statsout` named, for instance, `output_pdf`, and pass "pdf" as the `format` into the `output` function.

❸ Now you can simply call the output function in the same way as any other function. The `output_function` variable is a reference to the appropriate function from the `statsout` module.

Did you see the bug in the previous example? This is a very loose coupling of strings and functions, and there is no error checking. What happens if the user passes in a format that doesn't have a corresponding function defined in `statsout`? Well, `getattr` will return `None`, which will be assigned to `output_function` instead of a valid function, and the next line that attempts to call that function will crash and raise an exception. That's bad.

Luckily, `getattr` takes an optional third argument, a default value.

Example 4.13. `getattr` Default Values

```
import statsout

def output(data, format="text"):
    output_function = getattr(statsout, "output_%s" % format, statsout.output_text)
    return output_function(data) ❶
```

❶ This function call is guaranteed to work, because you added a third argument to the call to `getattr`. The third argument is a default value that is returned if the attribute or method specified by the second argument wasn't found.

As you can see, `getattr` is quite powerful. It is the heart of introspection, and you'll see even more powerful examples of it in later chapters.

4.5. Filtering Lists

As you know, Python has powerful capabilities for mapping lists into other lists, via list comprehensions (Section 3.6, Mapping Lists). This can be combined with a filtering mechanism, where some elements in the list are mapped while others are skipped entirely.

Here is the list filtering syntax:

```
[mapping-expression for element in source-list if filter-expression]
```

This is an extension of the list comprehensions that you know and love. The first two thirds are the same; the last part, starting with the `if`, is the filter expression. A filter expression can be any expression that evaluates true or false (which in Python can be almost anything). Any element for which the filter expression evaluates true will be included in the mapping. All other elements are ignored, so they are never put through the mapping expression and are not included in the output list.

Example 4.14. Introducing List Filtering

```
>>> li = ["a", "mpilgrim", "foo", "b", "c", "b", "d", "d"]
>>> [elem for elem in li if len(elem) > 1]       ❶
['mpilgrim', 'foo']
>>> [elem for elem in li if elem != "b"]         ❷
```

```
['a', 'mpilgrim', 'foo', 'c', 'd', 'd']
>>> [elem for elem in li if li.count(elem) == 1] ❸
['a', 'mpilgrim', 'foo', 'c']
```

❶ The mapping expression here is simple (it just returns the value of each element), so concentrate on the filter expression. As Python loops through the list, it runs each element through the filter expression. If the filter expression is true, the element is mapped and the result of the mapping expression is included in the returned list. Here, you are filtering out all the one−character strings, so you're left with a list of all the longer strings.

❷ Here, you are filtering out a specific value, `b`. Note that this filters all occurrences of `b`, since each time it comes up, the filter expression will be false.

❸ `count` is a list method that returns the number of times a value occurs in a list. You might think that this filter would eliminate duplicates from a list, returning a list containing only one copy of each value in the original list. But it doesn't, because values that appear twice in the original list (in this case, `b` and `d`) are excluded completely. There are ways of eliminating duplicates from a list, but filtering is not the solution.

Let's get back to this line from `apihelper.py`:

```
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
```

This looks complicated, and it is complicated, but the basic structure is the same. The whole filter expression returns a list, which is assigned to the `methodList` variable. The first half of the expression is the list mapping part. The mapping expression is an identity expression, which it returns the value of each element. `dir(object)` returns a list of `object`'s attributes and methods −− that's the list you're mapping. So the only new part is the filter expression after the `if`.

The filter expression looks scary, but it's not. You already know about `callable`, `getattr`, and `in`. As you saw in the previous section, the expression `getattr(object, method)` returns a function object if `object` is a module and `method` is the name of a function in that module.

So this expression takes an object (named `object`). Then it gets a list of the names of the object's attributes, methods, functions, and a few other things. Then it filters that list to weed out all the stuff that you don't care about. You do the weeding out by taking the name of each attribute/method/function and getting a reference to the real thing, via the `getattr` function. Then you check to see if that object is callable, which will be any methods and functions, both built−in (like the `pop` method of a list) and user−defined (like the `buildConnectionString` function of the `odbchelper` module). You don't care about other attributes, like the `__name__` attribute that's built in to every module.

Further Reading on Filtering Lists

- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses another way to filter lists using the built−in `filter` function (http://www.python.org/doc/current/tut/node7.html#SECTION007130000000000000000).

4.6. The Peculiar Nature of and and or

In Python, `and` and `or` perform boolean logic as you would expect, but they do not return boolean values; instead, they return one of the actual values they are comparing.

Example 4.15. Introducing and

```
>>> 'a' and 'b'          ❶
```

```
'b'
>>> '' and 'b'          ❷
''
>>> 'a' and 'b' and 'c' ❸
'c'
```

❶ When using `and`, values are evaluated in a boolean context from left to right. `0`, `''`, `[]`, `()`, `{}`, and `None` are false in a boolean context; everything else is true. Well, almost everything. By default, instances of classes are true in a boolean context, but you can define special methods in your class to make an instance evaluate to false. You'll learn all about classes and special methods in Chapter 5. If all values are true in a boolean context, `and` returns the last value. In this case, `and` evaluates `'a'`, which is true, then `'b'`, which is true, and returns `'b'`.

❷ If any value is false in a boolean context, `and` returns the first false value. In this case, `''` is the first false value.

❸ All values are true, so `and` returns the last value, `'c'`.

Example 4.16. Introducing `or`

```
>>> 'a' or 'b'          ❶
'a'
>>> '' or 'b'           ❷
'b'
>>> '' or [] or {}      ❸
{}
>>> def sidefx():
...     print "in sidefx()"
...     return 1
>>> 'a' or sidefx()     ❹
'a'
```

❶ When using `or`, values are evaluated in a boolean context from left to right, just like `and`. If any value is true, `or` returns that value immediately. In this case, `'a'` is the first true value.

❷ `or` evaluates `''`, which is false, then `'b'`, which is true, and returns `'b'`.

❸ If all values are false, `or` returns the last value. `or` evaluates `''`, which is false, then `[]`, which is false, then `{}`, which is false, and returns `{}`.

❹ Note that `or` evaluates values only until it finds one that is true in a boolean context, and then it ignores the rest. This distinction is important if some values can have side effects. Here, the function `sidefx` is never called, because `or` evaluates `'a'`, which is true, and returns `'a'` immediately.

If you're a C hacker, you are certainly familiar with the `bool ? a : b` expression, which evaluates to `a` if `bool` is true, and `b` otherwise. Because of the way `and` and `or` work in Python, you can accomplish the same thing.

4.6.1. Using the and−or Trick

Example 4.17. Introducing the `and−or` Trick

```
>>> a = "first"
>>> b = "second"
>>> 1 and a or b ❶
'first'
>>> 0 and a or b ❷
'second'
```

❶

This syntax looks similar to the *bool* ? a : b expression in C. The entire expression is evaluated from left to right, so the and is evaluated first. 1 and 'first' evalutes to 'first', then 'first' or 'second' evalutes to 'first'.

❷ 0 and 'first' evalutes to False, and then 0 or 'second' evaluates to 'second'.

However, since this Python expression is simply boolean logic, and not a special construct of the language, there is one extremely important difference between this and−or trick in Python and the *bool* ? a : b syntax in C. If the value of a is false, the expression will not work as you would expect it to. (Can you tell I was bitten by this? More than once?)

Example 4.18. When the and−or Trick Fails

```
>>> a = ""
>>> b = "second"
>>> 1 and a or b         ❶
'second'
```

❶ Since a is an empty string, which Python considers false in a boolean context, 1 and '' evalutes to '', and then '' or 'second' evalutes to 'second'. Oops! That's not what you wanted.

The and−or trick, *bool* and a or b, will not work like the C expression *bool* ? a : b when a is false in a boolean context.

The real trick behind the and−or trick, then, is to make sure that the value of a is never false. One common way of doing this is to turn a into [a] and b into [b], then taking the first element of the returned list, which will be either a or b.

Example 4.19. Using the and−or Trick Safely

```
>>> a = ""
>>> b = "second"
>>> (1 and [a] or [b])[0] ❶
''
```

❶ Since [a] is a non−empty list, it is never false. Even if a is 0 or '' or some other false value, the list [a] is true because it has one element.

By now, this trick may seem like more trouble than it's worth. You could, after all, accomplish the same thing with an if statement, so why go through all this fuss? Well, in many cases, you are choosing between two constant values, so you can use the simpler syntax and not worry, because you know that the a value will always be true. And even if you need to use the more complicated safe form, there are good reasons to do so. For example, there are some cases in Python where if statements are not allowed, such as in lambda functions.

Further Reading on the and−or Trick

- Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/) discusses alternatives to the and−or trick (http://www.activestate.com/ASPN/Python/Cookbook/Recipe/52310).

4.7. Using lambda Functions

Python supports an interesting syntax that lets you define one−line mini−functions on the fly. Borrowed from Lisp, these so−called lambda functions can be used anywhere a function is required.

Example 4.20. Introducing `lambda` Functions

```
>>> def f(x):
...     return x*2
...
>>> f(3)
6
>>> g = lambda x: x*2  ❶
>>> g(3)
6
>>> (lambda x: x*2)(3) ❷
6
```

❶ This is a `lambda` function that accomplishes the same thing as the normal function above it. Note the abbreviated syntax here: there are no parentheses around the argument list, and the `return` keyword is missing (it is implied, since the entire function can only be one expression). Also, the function has no name, but it can be called through the variable it is assigned to.

❷ You can use a `lambda` function without even assigning it to a variable. This may not be the most useful thing in the world, but it just goes to show that a lambda is just an in−line function.

To generalize, a `lambda` function is a function that takes any number of arguments (including optional arguments) and returns the value of a single expression. `lambda` functions can not contain commands, and they can not contain more than one expression. Don't try to squeeze too much into a `lambda` function; if you need something more complex, define a normal function instead and make it as long as you want.

`lambda` functions are a matter of style. Using them is never required; anywhere you could use them, you could define a separate normal function and use that instead. I use them in places where I want to encapsulate specific, non−reusable code without littering my code with a lot of little one−line functions.

4.7.1. Real−World `lambda` Functions

Here are the `lambda` functions in `apihelper.py`:

```
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
```

Notice that this uses the simple form of the `and−or` trick, which is okay, because a `lambda` function is always true in a boolean context. (That doesn't mean that a `lambda` function can't return a false value. The function is always true; its return value could be anything.)

Also notice that you're using the `split` function with no arguments. You've already seen it used with one or two arguments, but without any arguments it splits on whitespace.

Example 4.21. `split` With No Arguments

```
>>> s = "this   is\na\ttest"  ❶
>>> print s
this   is
a       test
>>> print s.split()           ❷
['this', 'is', 'a', 'test']
>>> print " ".join(s.split()) ❸
'this is a test'
```

❶

This is a multiline string, defined by escape characters instead of triple quotes. `\n` is a carriage return, and `\t` is a tab character.

❷ `split` without any arguments splits on whitespace. So three spaces, a carriage return, and a tab character are all the same.

❸ You can normalize whitespace by splitting a string with `split` and then rejoining it with `join`, using a single space as a delimiter. This is what the `info` function does to collapse multi−line `doc string`s into a single line.

So what is the `info` function actually doing with these `lambda` functions, `split`s, and `and−or` tricks?

```
processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
```

`processFunc` is now a function, but which function it is depends on the value of the `collapse` variable. If `collapse` is true, `processFunc(`*`string`*`)` will collapse whitespace; otherwise, `processFunc(`*`string`*`)` will return its argument unchanged.

To do this in a less robust language, like Visual Basic, you would probably create a function that took a string and a *`collapse`* argument and used an `if` statement to decide whether to collapse the whitespace or not, then returned the appropriate value. This would be inefficient, because the function would need to handle every possible case. Every time you called it, it would need to decide whether to collapse whitespace before it could give you what you wanted. In Python, you can take that decision logic out of the function and define a `lambda` function that is custom−tailored to give you exactly (and only) what you want. This is more efficient, more elegant, and less prone to those nasty oh−I−thought−those−arguments−were−reversed kinds of errors.

Further Reading on `lambda` Functions

- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) discusses using `lambda` to call functions indirectly (http://www.faqts.com/knowledge−base/view.phtml/aid/6081/fid/241).
- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) shows how to access outside variables from inside a `lambda` function (http://www.python.org/doc/current/tut/node6.html#SECTION006740000000000000000). (PEP 227 (http://python.sourceforge.net/peps/pep−0227.html) explains how this will change in future versions of Python.)
- *The Whole Python FAQ* (http://www.python.org/doc/FAQ.html) has examples of obfuscated one−liners using `lambda` (http://www.python.org/cgi−bin/faqw.py?query=4.15&querytype=simple&casefold=yes&req=search).

4.8. Putting It All Together

The last line of code, the only one you haven't deconstructed yet, is the one that does all the work. But by now the work is easy, because everything you need is already set up just the way you need it. All the dominoes are in place; it's time to knock them down.

This is the meat of `apihelper.py`:

```
print "\n".join(["%s %s" %
                  (method.ljust(spacing),
                   processFunc(str(getattr(object, method).__doc__)))
                 for method in methodList])
```

Note that this is one command, split over multiple lines, but it doesn't use the line continuation character (\). Remember when I said that some expressions can be split into multiple lines without using a backslash? A list

comprehension is one of those expressions, since the entire expression is contained in square brackets.

Now, let's take it from the end and work backwards. The

```
for method in methodList
```

shows that this is a list comprehension. As you know, `methodList` is a list of all the methods you care about in `object`. So you're looping through that list with `method`.

Example 4.22. Getting a `doc string` Dynamically

```
>>> import odbchelper
>>> object = odbchelper               ❶
>>> method = 'buildConnectionString'  ❷
>>> getattr(object, method)           ❸
<function buildConnectionString at 010D6D74>
>>> print getattr(object, method).__doc__ ❹
Build a connection string from a dictionary of parameters.

    Returns string.
```

❶ In the `info` function, `object` is the object you're getting help on, passed in as an argument.

❷ As you're looping through `methodList`, `method` is the name of the current method.

❸ Using the `getattr` function, you're getting a reference to the *`method`* function in the *`object`* module.

❹ Now, printing the actual `doc string` of the method is easy.

The next piece of the puzzle is the use of `str` around the `doc string`. As you may recall, `str` is a built−in function that coerces data into a string. But a `doc string` is always a string, so why bother with the `str` function? The answer is that not every function has a `doc string`, and if it doesn't, its `__doc__` attribute is `None`.

Example 4.23. Why Use `str` on a `doc string`?

```
>>> >>> def foo(): print 2
>>> >>> foo()
2
>>> >>> foo.__doc__      ❶
>>> foo.__doc__ == None ❷
True
>>> str(foo.__doc__)    ❸
'None'
```

❶ You can easily define a function that has no `doc string`, so its `__doc__` attribute is `None`. Confusingly, if you evaluate the `__doc__` attribute directly, the Python IDE prints nothing at all, which makes sense if you think about it, but is still unhelpful.

❷ You can verify that the value of the `__doc__` attribute is actually `None` by comparing it directly.

❸ The `str` function takes the null value and returns a string representation of it, `'None'`.

In SQL, you must use `IS NULL` instead of `= NULL` to compare a null value. In Python, you can use either `== None` or `is None`, but `is None` is faster.

Now that you are guaranteed to have a string, you can pass the string to `processFunc`, which you have already defined as a function that either does or doesn't collapse whitespace. Now you see why it was important to use `str` to convert a `None` value into a string representation. `processFunc` is assuming a string argument and calling its `split` method, which would crash if you passed it `None` because `None` doesn't have a `split` method.

Stepping back even further, you see that you're using string formatting again to concatenate the return value of `processFunc` with the return value of `method`'s `ljust` method. This is a new string method that you haven't seen before.

Example 4.24. Introducing `ljust`

```
>>> s = 'buildConnectionString'
>>> s.ljust(30) ❶
'buildConnectionString         '
>>> s.ljust(20) ❷
'buildConnectionString'
```

❶ `ljust` pads the string with spaces to the given length. This is what the `info` function uses to make two columns of output and line up all the `doc string`s in the second column.

❷ If the given length is smaller than the length of the string, `ljust` will simply return the string unchanged. It never truncates the string.

You're almost finished. Given the padded method name from the `ljust` method and the (possibly collapsed) `doc string` from the call to `processFunc`, you concatenate the two and get a single string. Since you're mapping `methodList`, you end up with a list of strings. Using the `join` method of the string `"\n"`, you join this list into a single string, with each element of the list on a separate line, and print the result.

Example 4.25. Printing a List

```
>>> li = ['a', 'b', 'c']
>>> print "\n".join(li) ❶
a
b
c
```

❶ This is also a useful debugging trick when you're working with lists. And in Python, you're always working with lists.

That's the last piece of the puzzle. You should now understand this code.

```
    print "\n".join(["%s %s" %
                      (method.ljust(spacing),
                       processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])
```

4.9. Summary

The `apihelper.py` program and its output should now make perfect sense.

```
def info(object, spacing=10, collapse=1):
    """Print methods and doc strings.

    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                      (method.ljust(spacing),
                       processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])

if __name__ == "__main__":
```

```
    print info.__doc__
```

Here is the output of `apihelper.py`:

```
>>> from apihelper import info
>>> li = []
>>> info(li)
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences of value
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1, 0, 1
```

Before diving into the next chapter, make sure you're comfortable doing all of these things:

- Defining and calling functions with optional and named arguments
- Using `str` to coerce any arbitrary value into a string representation
- Using `getattr` to get references to functions and other attributes dynamically
- Extending the list comprehension syntax to do list filtering
- Recognizing the `and-or` trick and using it safely
- Defining `lambda` functions
- Assigning functions to variables and calling the function by referencing the variable. I can't emphasize this enough, because this mode of thought is vital to advancing your understanding of Python. You'll see more complex applications of this concept throughout this book.

Chapter 5. Objects and Object−Orientation

This chapter, and pretty much every chapter after this, deals with object−oriented Python programming.

5.1. Diving In

Here is a complete, working Python program. Read the `doc string`s of the module, the classes, and the functions to get an overview of what this program does and how it works. As usual, don't worry about the stuff you don't understand; that's what the rest of the chapter is for.

Example 5.1. `fileinfo.py`

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
"""Framework for getting filetype−specific metadata.

Instantiate appropriate class with filename.  Returned object acts like a
dictionary, with key−value pairs for each piece of metadata.
    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])

Or use listDirectory function to get info on all files in a directory.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...

Framework can be extended by adding classes for particular file types, e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Each class is completely responsible for
parsing its files appropriately; see MP3FileInfo for example.
"""
import os
import sys
from UserDict import UserDict

def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()

class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename

class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}

    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
        self.clear()
        try:
```

```
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass

    def __setitem__(self, key, item):
        if key == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f)
               for f in fileList
                if os.path.splitext(f)[1] in fileExtList]
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]

if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]): ❶
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
        print
```

❶ This program's output depends on the files on your hard drive. To get meaningful output, you'll need to change the directory path to point to a directory of MP3 files on your own machine.

This is the output I got on my machine. Your output will be different, unless, by some startling coincidence, you share my exact taste in music.

```
album=
artist=Ghost in the Machine
title=A Time Long Forgotten (Concept
genre=31
name=/music/_singles/a_time_long_forgotten_con.mp3
year=1999
comment=http://mp3.com/ghostmachine

album=Rave Mix
artist=***DJ MARY-JANE***
title=HELLRAISER****Trance from Hell
genre=31
name=/music/_singles/hellraiser.mp3
year=2000
comment=http://mp3.com/DJMARYJANE

album=Rave Mix
artist=***DJ MARY-JANE***
title=KAIRO****THE BEST GOA
genre=31
name=/music/_singles/kairo.mp3
year=2000
```

```
comment=http://mp3.com/DJMARYJANE

album=Journeys
artist=Masters of Balance
title=Long Way Home
genre=31
name=/music/_singles/long_way_home1.mp3
year=2000
comment=http://mp3.com/MastersofBalan

album=
artist=The Cynic Project
title=Sidewinder
genre=18
name=/music/_singles/sidewinder.mp3
year=2000
comment=http://mp3.com/cynicproject

album=Digitosis@128k
artist=VXpanded
title=Spinning
genre=255
name=/music/_singles/spinning.mp3
year=2000
comment=http://mp3.com/artists/95/vxp
```

5.2. Importing Modules Using `from module import`

Python has two ways of importing modules. Both are useful, and you should know when to use each. One way, `import module`, you've already seen in Section 2.4, Everything Is an Object . The other way accomplishes the same thing, but it has subtle and important differences.

Here is the basic `from module import` syntax:

```
from UserDict import UserDict
```

This is similar to the `import module` syntax that you know and love, but with an important difference: the attributes and methods of the imported module `types` are imported directly into the local namespace, so they are available directly, without qualification by module name. You can import individual items or use `from module import *` to import everything.

`from module import *` in Python is like `use module` in Perl; `import module` in Python is like `require module` in Perl.

`from module import *` in Python is like `import module.*` in Java; `import module` in Python is like `import module` in Java.

Example 5.2. `import module` vs. `from module import`

```
>>> import types
>>> types.FunctionType             ❶
<type 'function'>
>>> FunctionType                   ❷
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'FunctionType'
>>> from types import FunctionType ❸
```

```
>>> FunctionType                    ❹
<type 'function'>
```

❶ The `types` module contains no methods; it just has attributes for each Python object type. Note that the attribute, `FunctionType`, must be qualified by the module name, `types`.

❷ `FunctionType` by itself has not been defined in this namespace; it exists only in the context of `types`.

❸ This syntax imports the attribute `FunctionType` from the `types` module directly into the local namespace.

❹ Now `FunctionType` can be accessed directly, without reference to `types`.

When should you use `from` *module* `import`?

- If you will be accessing attributes and methods often and don't want to type the module name over and over, use `from` *module* `import`.
- If you want to selectively import some attributes and methods but not others, use `from` *module* `import`.
- If the module contains attributes or functions with the same name as ones in your module, you must use `import` *module* to avoid name conflicts.

Other than that, it's just a matter of style, and you will see Python code written both ways.

Use `from module import *` sparingly, because it makes it difficult to determine where a particular function or attribute came from, and that makes debugging and refactoring more difficult.

Further Reading on Module Importing Techniques

- eff−bot (http://www.effbot.org/guides/) has more to say on `import` *module* *vs.* `from` *module* `import` (http://www.effbot.org/guides/import−confusion.htm).
- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses advanced import techniques, including `from` *module* `import *` (http://www.python.org/doc/current/tut/node8.html#SECTION008410000000000000000).

5.3. Defining Classes

Python is fully object−oriented: you can define your own classes, inherit from your own or built−in classes, and instantiate the classes you've defined.

Defining a class in Python is simple. As with functions, there is no separate interface definition. Just define the class and start coding. A Python class starts with the reserved word `class`, followed by the class name. Technically, that's all that's required, since a class doesn't need to inherit from any other class.

Example 5.3. The Simplest Python Class

```
class Loaf: ❶
    pass    ❷ ❸
```

❶ The name of this class is `Loaf`, and it doesn't inherit from any other class. Class names are usually capitalized, `EachWordLikeThis`, but this is only a convention, not a requirement.

❷ This class doesn't define any methods or attributes, but syntactically, there needs to be something in the definition, so you use `pass`. This is a Python reserved word that just means "move along, nothing to see here". It's a statement that does nothing, and it's a good placeholder when you're stubbing out functions or classes.

❸ You probably guessed this, but everything in a class is indented, just like the code within a function, `if` statement, `for` loop, and so forth. The first thing not indented is not in the class.

The `pass` statement in Python is like an empty set of braces (`{}`) in Java or C.

Of course, realistically, most classes will be inherited from other classes, and they will define their own class methods and attributes. But as you've just seen, there is nothing that a class absolutely must have, other than a name. In particular, C++ programmers may find it odd that Python classes don't have explicit constructors and destructors. Python classes do have something similar to a constructor: the `__init__` method.

Example 5.4. Defining the `FileInfo` Class

```
from UserDict import UserDict

class FileInfo(UserDict): ❶
```

❶ In Python, the ancestor of a class is simply listed in parentheses immediately after the class name. So the `FileInfo` class is inherited from the `UserDict` class (which was imported from the `UserDict` module). `UserDict` is a class that acts like a dictionary, allowing you to essentially subclass the dictionary datatype and add your own behavior. (There are similar classes `UserList` and `UserString` which allow you to subclass lists and strings.) There is a bit of black magic behind this, which you will demystify later in this chapter when you explore the `UserDict` class in more depth.

In Python, the ancestor of a class is simply listed in parentheses immediately after the class name. There is no special keyword like `extends` in Java.

Python supports multiple inheritance. In the parentheses following the class name, you can list as many ancestor classes as you like, separated by commas.

5.3.1. Initializing and Coding Classes

This example shows the initialization of the `FileInfo` class using the `__init__` method.

Example 5.5. Initializing the `FileInfo` Class

```
class FileInfo(UserDict):
    "store file metadata"              ❶
    def __init__(self, filename=None): ❷ ❸ ❹
```

❶ Classes can (and should) have `doc string`s too, just like modules and functions.

❷ `__init__` is called immediately after an instance of the class is created. It would be tempting but incorrect to call this the constructor of the class. It's tempting, because it looks like a constructor (by convention, `__init__` is the first method defined for the class), acts like one (it's the first piece of code executed in a newly created instance of the class), and even sounds like one ("init" certainly suggests a constructor−ish nature). Incorrect, because the object has already been constructed by the time `__init__` is called, and you already have a valid reference to the new instance of the class. But `__init__` is the closest thing you're going to get to a constructor in Python, and it fills much the same role.

❸ The first argument of every class method, including `__init__`, is always a reference to the current instance of the class. By convention, this argument is always named `self`. In the `__init__` method, `self` refers to the newly created object; in other class methods, it refers to the instance whose method was called. Although you need to specify `self` explicitly when defining the method, you do *not* specify it when calling the method; Python will add it for you automatically.

4. `__init__` methods can take any number of arguments, and just like functions, the arguments can be defined with default values, making them optional to the caller. In this case, `filename` has a default value of `None`, which is the Python null value.

By convention, the first argument of any Python class method (the reference to the current instance) is called `self`. This argument fills the role of the reserved word `this` in C++ or Java, but `self` is not a reserved word in Python, merely a naming convention. Nonetheless, please don't call it anything but `self`; this is a very strong convention.

Example 5.6. Coding the `FileInfo` Class

```
class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)        (1)
        self["name"] = filename        (2)
                                       (3)
```

1. Some pseudo−object−oriented languages like Powerbuilder have a concept of "extending" constructors and other events, where the ancestor's method is called automatically before the descendant's method is executed. Python does not do this; you must always explicitly call the appropriate method in the ancestor class.
2. I told you that this class acts like a dictionary, and here is the first sign of it. You're assigning the argument `filename` as the value of this object's `name` key.
3. Note that the `__init__` method never returns a value.

5.3.2. Knowing When to Use `self` and `__init__`

When defining your class methods, you *must* explicitly list `self` as the first argument for each method, including `__init__`. When you call a method of an ancestor class from within your class, you *must* include the `self` argument. But when you call your class method from outside, you do not specify anything for the `self` argument; you skip it entirely, and Python automatically adds the instance reference for you. I am aware that this is confusing at first; it's not really inconsistent, but it may appear inconsistent because it relies on a distinction (between bound and unbound methods) that you don't know about yet.

Whew. I realize that's a lot to absorb, but you'll get the hang of it. All Python classes work the same way, so once you learn one, you've learned them all. If you forget everything else, remember this one thing, because I promise it will trip you up:

`__init__` methods are optional, but when you define one, you must remember to explicitly call the ancestor's `__init__` method (if it defines one). This is more generally true: whenever a descendant wants to extend the behavior of the ancestor, the descendant method must explicitly call the ancestor method at the proper time, with the proper arguments.

Further Reading on Python Classes

- *Learning to Program* (http://www.freenetpages.co.uk/hp/alan.gauld/) has a gentler introduction to classes (http://www.freenetpages.co.uk/hp/alan.gauld/tutclass.htm).
- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) shows how to use classes to model compound datatypes (http://www.ibiblio.org/obp/thinkCSpy/chap12.htm).
- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) has an in−depth look at classes, namespaces, and inheritance (http://www.python.org/doc/current/tut/node11.html).
- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) answers common questions about classes (http://www.faqts.com/knowledge−base/index.phtml/fid/242).

5.4. Instantiating Classes

Instantiating classes in Python is straightforward. To instantiate a class, simply call the class as if it were a function, passing the arguments that the `__init__` method defines. The return value will be the newly created object.

Example 5.7. Creating a `FileInfo` Instance

```
>>> import fileinfo
>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3") (1)
>>> f.__class__                                        (2)
<class fileinfo.FileInfo at 010EC204>
>>> f.__doc__                                          (3)
'store file metadata'
>>> f                                                  (4)
{'name': '/music/_singles/kairo.mp3'}
```

(1) You are creating an instance of the `FileInfo` class (defined in the `fileinfo` module) and assigning the newly created instance to the variable `f`. You are passing one parameter, `/music/_singles/kairo.mp3`, which will end up as the `filename` argument in `FileInfo`'s `__init__` method.

(2) Every class instance has a built−in attribute, `__class__`, which is the object's class. (Note that the representation of this includes the physical address of the instance on my machine; your representation will be different.) Java programmers may be familiar with the `Class` class, which contains methods like `getName` and `getSuperclass` to get metadata information about an object. In Python, this kind of metadata is available directly on the object itself through attributes like `__class__`, `__name__`, and `__bases__`.

(3) You can access the instance's `doc string` just as with a function or a module. All instances of a class share the same `doc string`.

(4) Remember when the `__init__` method assigned its `filename` argument to `self["name"]`? Well, here's the result. The arguments you pass when you create the class instance get sent right along to the `__init__` method (along with the object reference, `self`, which Python adds for free).

In Python, simply call a class as if it were a function to create a new instance of the class. There is no explicit `new` operator like C++ or Java.

5.4.1. Garbage Collection

If creating new instances is easy, destroying them is even easier. In general, there is no need to explicitly free instances, because they are freed automatically when the variables assigned to them go out of scope. Memory leaks are rare in Python.

Example 5.8. Trying to Implement a Memory Leak

```
>>> def leakmem():
...     f = fileinfo.FileInfo('/music/_singles/kairo.mp3') (1)
...
>>> for i in range(100):
...     leakmem()                                           (2)
```

(1) Every time the `leakmem` function is called, you are creating an instance of `FileInfo` and assigning it to the variable `f`, which is a local variable within the function. Then the function ends without ever freeing `f`, so you would expect a memory leak, but you would be wrong. When the function ends, the local variable `f` goes out of scope. At this point, there are no longer any references to the newly created instance of `FileInfo` (since you never assigned it to anything other than `f`), so Python destroys the

instance for us.

❷ No matter how many times you call the `leakmem` function, it will never leak memory, because every time, Python will destroy the newly created `FileInfo` class before returning from `leakmem`.

The technical term for this form of garbage collection is "reference counting". Python keeps a list of references to every instance created. In the above example, there was only one reference to the `FileInfo` instance: the local variable `f`. When the function ends, the variable `f` goes out of scope, so the reference count drops to `0`, and Python destroys the instance automatically.

In previous versions of Python, there were situations where reference counting failed, and Python couldn't clean up after you. If you created two instances that referenced each other (for instance, a doubly−linked list, where each node has a pointer to the previous and next node in the list), neither instance would ever be destroyed automatically because Python (correctly) believed that there is always a reference to each instance. Python 2.0 has an additional form of garbage collection called "mark−and−sweep" which is smart enough to notice this virtual gridlock and clean up circular references correctly.

As a former philosophy major, it disturbs me to think that things disappear when no one is looking at them, but that's exactly what happens in Python. In general, you can simply forget about memory management and let Python clean up after you.

Further Reading on Garbage Collection

- *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes built−in attributes like `__class__` (http://www.python.org/doc/current/lib/specialattrs.html).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the `gc` module (http://www.python.org/doc/current/lib/module−gc.html), which gives you low−level control over Python's garbage collection.

5.5. Exploring `UserDict`: A Wrapper Class

As you've seen, `FileInfo` is a class that acts like a dictionary. To explore this further, let's look at the `UserDict` class in the `UserDict` module, which is the ancestor of the `FileInfo` class. This is nothing special; the class is written in Python and stored in a `.py` file, just like any other Python code. In particular, it's stored in the `lib` directory in your Python installation.

In the ActivePython IDE on Windows, you can quickly open any module in your library path by selecting File−>Locate... (**Ctrl−L**).

Example 5.9. Defining the `UserDict` Class

```
class UserDict:                                ❶
    def __init__(self, dict=None):             ❷
        self.data = {}                         ❸
        if dict is not None: self.update(dict) ❹ ❺
```

❶ Note that `UserDict` is a base class, not inherited from any other class.

❷ This is the `__init__` method that you overrode in the `FileInfo` class. Note that the argument list in this ancestor class is different than the descendant. That's okay; each subclass can have its own set of arguments, as long as it calls the ancestor with the correct arguments. Here the ancestor class has a way to define initial values (by passing a dictionary in the `dict` argument) which the `FileInfo` does not use.

❸ Python supports data attributes (called "instance variables" in Java and Powerbuilder, and "member variables" in C++). Data attributes are pieces of data held by a specific instance of a class. In this case, each instance of `UserDict` will have a data attribute `data`. To reference this attribute from code outside the class, you qualify it with the instance name, *`instance`*`.data`, in the same way that you qualify a function with its module name. To reference a data attribute from within the class, you use `self` as the qualifier. By convention, all data attributes are initialized to reasonable values in the `__init__` method. However, this is not required, since data attributes, like local variables, spring into existence when they are first assigned a value.

❹ The `update` method is a dictionary duplicator: it copies all the keys and values from one dictionary to another. This does *not* clear the target dictionary first; if the target dictionary already has some keys, the ones from the source dictionary will be overwritten, but others will be left untouched. Think of `update` as a merge function, not a copy function.

❺ This is a syntax you may not have seen before (I haven't used it in the examples in this book). It's an `if` statement, but instead of having an indented block starting on the next line, there is just a single statement on the same line, after the colon. This is perfectly legal syntax, which is just a shortcut you can use when you have only one statement in a block. (It's like specifying a single statement without braces in C++.) You can use this syntax, or you can have indented code on subsequent lines, but you can't do both for the same block.

Java and Powerbuilder support function overloading by argument list, *i.e.* one class can have multiple methods with the same name but a different number of arguments, or arguments of different types. Other languages (most notably PL/SQL) even support function overloading by argument name; *i.e.* one class can have multiple methods with the same name and the same number of arguments of the same type but different argument names. Python supports neither of these; it has no form of function overloading whatsoever. Methods are defined solely by their name, and there can be only one method per class with a given name. So if a descendant class has an `__init__` method, it *always* overrides the ancestor `__init__` method, even if the descendant defines it with a different argument list. And the same rule applies to any other method.

Guido, the original author of Python, explains method overriding this way: "Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class, may in fact end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)" If that doesn't make sense to you (it confuses the hell out of me), feel free to ignore it. I just thought I'd pass it along.

Always assign an initial value to all of an instance's data attributes in the `__init__` method. It will save you hours of debugging later, tracking down `AttributeError` exceptions because you're referencing uninitialized (and therefore non−existent) attributes.

Example 5.10. `UserDict` Normal Methods

```
    def clear(self): self.data.clear()          ❶
    def copy(self):                             ❷
        if self.__class__ is UserDict:          ❸
            return UserDict(self.data)
        import copy                             ❹
        return copy.copy(self)
    def keys(self): return self.data.keys()     ❺
    def items(self): return self.data.items()
    def values(self): return self.data.values()
```

❶ `clear` is a normal class method; it is publicly available to be called by anyone at any time. Notice that `clear`, like all class methods, has `self` as its first argument. (Remember that you don't include `self` when you call the method; it's something that Python adds for you.) Also note the basic technique of this wrapper class: store a real dictionary (`data`) as a data attribute, define all the methods that a real dictionary has, and have each class method redirect to the corresponding method on the real dictionary. (In case you'd forgotten, a dictionary's

`clear` method deletes all of its keys and their associated values.)

❷ The `copy` method of a real dictionary returns a new dictionary that is an exact duplicate of the original (all the same key−value pairs). But `UserDict` can't simply redirect to `self.data.copy`, because that method returns a real dictionary, and what you want is to return a new instance that is the same class as `self`.

❸ You use the `__class__` attribute to see if `self` is a `UserDict`; if so, you're golden, because you know how to copy a `UserDict`: just create a new `UserDict` and give it the real dictionary that you've squirreled away in `self.data`. Then you immediately return the new `UserDict` you don't even get to the `import copy` on the next line.

❹ If `self.__class__` is not `UserDict`, then `self` must be some subclass of `UserDict` (like maybe `FileInfo`), in which case life gets trickier. `UserDict` doesn't know how to make an exact copy of one of its descendants; there could, for instance, be other data attributes defined in the subclass, so you would need to iterate through them and make sure to copy all of them. Luckily, Python comes with a module to do exactly this, and it's called `copy`. I won't go into the details here (though it's a wicked cool module, if you're ever inclined to dive into it on your own). Suffice it to say that `copy` can copy arbitrary Python objects, and that's how you're using it here.

❺ The rest of the methods are straightforward, redirecting the calls to the built−in methods on `self.data`.

In versions of Python prior to 2.2, you could not directly subclass built−in datatypes like strings, lists, and dictionaries. To compensate for this, Python comes with wrapper classes that mimic the behavior of these built−in datatypes: `UserString`, `UserList`, and `UserDict`. Using a combination of normal and special methods, the `UserDict` class does an excellent imitation of a dictionary. In Python 2.2 and later, you can inherit classes directly from built−in datatypes like `dict`. An example of this is given in the examples that come with this book, in `fileinfo_fromdict.py`.

In Python, you can inherit directly from the `dict` built−in datatype, as shown in this example. There are three differences here compared to the `UserDict` version.

Example 5.11. Inheriting Directly from Built−In Datatype `dict`

```
class FileInfo(dict):                  ❶
    "store file metadata"
    def __init__(self, filename=None): ❷
        self["name"] = filename
```

❶ The first difference is that you don't need to import the `UserDict` module, since `dict` is a built−in datatype and is always available. The second is that you are inheriting from `dict` directly, instead of from `UserDict.UserDict`.

❷ The third difference is subtle but important. Because of the way `UserDict` works internally, it requires you to manually call its `__init__` method to properly initialize its internal data structures. `dict` does not work like this; it is not a wrapper, and it requires no explicit initialization.

Further Reading on `UserDict`

- *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the `UserDict` module (http://www.python.org/doc/current/lib/module−UserDict.html) and the `copy` module (http://www.python.org/doc/current/lib/module−copy.html).

5.6. Special Class Methods

In addition to normal class methods, there are a number of special methods that Python classes can define. Instead of being called directly by your code (like normal methods), special methods are called for you by Python in particular circumstances or when specific syntax is used.

As you saw in the previous section, normal methods go a long way towards wrapping a dictionary in a class. But normal methods alone are not enough, because there are a lot of things you can do with dictionaries besides call methods on them. For starters, you can get and set items with a syntax that doesn't include explicitly invoking methods. This is where special class methods come in: they provide a way to map non−method−calling syntax into method calls.

5.6.1. Getting and Setting Items

Example 5.12. The `__getitem__` Special Method

```
    def __getitem__(self, key): return self.data[key]

>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3")
>>> f
{'name':'/music/_singles/kairo.mp3'}
>>> f.__getitem__("name") ❶
'/music/_singles/kairo.mp3'
>>> f["name"]             ❷
'/music/_singles/kairo.mp3'
```

❶ The `__getitem__` special method looks simple enough. Like the normal methods `clear`, `keys`, and `values`, it just redirects to the dictionary to return its value. But how does it get called? Well, you can call `__getitem__` directly, but in practice you wouldn't actually do that; I'm just doing it here to show you how it works. The right way to use `__getitem__` is to get Python to call it for you.

❷ This looks just like the syntax you would use to get a dictionary value, and in fact it returns the value you would expect. But here's the missing link: under the covers, Python has converted this syntax to the method call `f.__getitem__("name")`. That's why `__getitem__` is a special class method; not only can you call it yourself, you can get Python to call it for you by using the right syntax.

Of course, Python has a `__setitem__` special method to go along with `__getitem__`, as shown in the next example.

Example 5.13. The `__setitem__` Special Method

```
    def __setitem__(self, key, item): self.data[key] = item

>>> f
{'name':'/music/_singles/kairo.mp3'}
>>> f.__setitem__("genre", 31) ❶
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':31}
>>> f["genre"] = 32            ❷
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':32}
```

❶ Like the `__getitem__` method, `__setitem__` simply redirects to the real dictionary `self.data` to do its work. And like `__getitem__`, you wouldn't ordinarily call it directly like this; Python calls `__setitem__` for you when you use the right syntax.

❷ This looks like regular dictionary syntax, except of course that `f` is really a class that's trying very hard to masquerade as a dictionary, and `__setitem__` is an essential part of that masquerade. This line of code actually calls `f.__setitem__("genre", 32)` under the covers.

`__setitem__` is a special class method because it gets called for you, but it's still a class method. Just as easily as the `__setitem__` method was defined in `UserDict`, you can redefine it in the descendant class to override the ancestor method. This allows you to define classes that act like dictionaries in some ways but define their own behavior above and beyond the built−in dictionary.

This concept is the basis of the entire framework you're studying in this chapter. Each file type can have a handler class that knows how to get metadata from a particular type of file. Once some attributes (like the file's name and location) are known, the handler class knows how to derive other attributes automatically. This is done by overriding the `__setitem__` method, checking for particular keys, and adding additional processing when they are found.

For example, `MP3FileInfo` is a descendant of `FileInfo`. When an `MP3FileInfo`'s `name` is set, it doesn't just set the `name` key (like the ancestor `FileInfo` does); it also looks in the file itself for MP3 tags and populates a whole set of keys. The next example shows how this works.

Example 5.14. Overriding `__setitem__` in `MP3FileInfo`

```
    def __setitem__(self, key, item):         ❶
        if key == "name" and item:            ❷
            self.__parse(item)                ❸
        FileInfo.__setitem__(self, key, item) ❹
```

❶ Notice that this `__setitem__` method is defined exactly the same way as the ancestor method. This is important, since Python will be calling the method for you, and it expects it to be defined with a certain number of arguments. (Technically speaking, the names of the arguments don't matter; only the number of arguments is important.)

❷ Here's the crux of the entire `MP3FileInfo` class: if you're assigning a value to the `name` key, you want to do something extra.

❸ The extra processing you do for `name`s is encapsulated in the `__parse` method. This is another class method defined in `MP3FileInfo`, and when you call it, you qualify it with `self`. Just calling `__parse` would look for a normal function defined outside the class, which is not what you want. Calling `self.__parse` will look for a class method defined within the class. This isn't anything new; you reference data attributes the same way.

❹ After doing this extra processing, you want to call the ancestor method. Remember that this is never done for you in Python; you must do it manually. Note that you're calling the immediate ancestor, `FileInfo`, even though it doesn't have a `__setitem__` method. That's okay, because Python will walk up the ancestor tree until it finds a class with the method you're calling, so this line of code will eventually find and call the `__setitem__` defined in `UserDict`.

When accessing data attributes within a class, you need to qualify the attribute name: `self.`*`attribute`*. When calling other methods within a class, you need to qualify the method name: `self.`*`method`*.

Example 5.15. Setting an `MP3FileInfo`'s `name`

```
>>> import fileinfo
>>> mp3file = fileinfo.MP3FileInfo()                   ❶
>>> mp3file
{'name':None}
>>> mp3file["name"] = "/music/_singles/kairo.mp3"      ❷
>>> mp3file
{'album': 'Rave Mix', 'artist': '***DJ MARY−JANE***', 'genre': 31,
'title': 'KAIRO****THE BEST GOA', 'name': '/music/_singles/kairo.mp3',
'year': '2000', 'comment': 'http://mp3.com/DJMARYJANE'}
>>> mp3file["name"] = "/music/_singles/sidewinder.mp3" ❸
>>> mp3file
{'album': '', 'artist': 'The Cynic Project', 'genre': 18, 'title': 'Sidewinder',
```

```
'name': '/music/_singles/sidewinder.mp3', 'year': '2000',
'comment': 'http://mp3.com/cynicproject'}
```

❶ First, you create an instance of `MP3FileInfo`, without passing it a filename. (You can get away with this because the `filename` argument of the `__init__` method is optional.) Since `MP3FileInfo` has no `__init__` method of its own, Python walks up the ancestor tree and finds the `__init__` method of `FileInfo`. This `__init__` method manually calls the `__init__` method of `UserDict` and then sets the `name` key to `filename`, which is `None`, since you didn't pass a filename. Thus, `mp3file` initially looks like a dictionary with one key, `name`, whose value is `None`.

❷ Now the real fun begins. Setting the `name` key of `mp3file` triggers the `__setitem__` method on `MP3FileInfo` (not `UserDict`), which notices that you're setting the `name` key with a real value and calls `self.__parse`. Although you haven't traced through the `__parse` method yet, you can see from the output that it sets several other keys: `album`, `artist`, `genre`, `title`, `year`, and `comment`.

❸ Modifying the `name` key will go through the same process again: Python calls `__setitem__`, which calls `self.__parse`, which sets all the other keys.

5.7. Advanced Special Class Methods

Python has more special methods than just `__getitem__` and `__setitem__`. Some of them let you emulate functionality that you may not even know about.

This example shows some of the other special methods in `UserDict`.

Example 5.16. More Special Methods in `UserDict`

```
    def __repr__(self): return repr(self.data)     ❶
    def __cmp__(self, dict):                        ❷
        if isinstance(dict, UserDict):
            return cmp(self.data, dict.data)
        else:
            return cmp(self.data, dict)
    def __len__(self): return len(self.data)        ❸
    def __delitem__(self, key): del self.data[key] ❹
```

❶ `__repr__` is a special method that is called when you call `repr(`*`instance`*`)`. The `repr` function is a built−in function that returns a string representation of an object. It works on any object, not just class instances. You're already intimately familiar with `repr` and you don't even know it. In the interactive window, when you type just a variable name and press the **ENTER** key, Python uses `repr` to display the variable's value. Go create a dictionary `d` with some data and then `print repr(d)` to see for yourself.

❷ `__cmp__` is called when you compare class instances. In general, you can compare any two Python objects, not just class instances, by using `==`. There are rules that define when built−in datatypes are considered equal; for instance, dictionaries are equal when they have all the same keys and values, and strings are equal when they are the same length and contain the same sequence of characters. For class instances, you can define the `__cmp__` method and code the comparison logic yourself, and then you can use `==` to compare instances of your class and Python will call your `__cmp__` special method for you.

❸ `__len__` is called when you call `len(`*`instance`*`)`. The `len` function is a built−in function that returns the length of an object. It works on any object that could reasonably be thought of as having a length. The `len` of a string is its number of characters; the `len` of a dictionary is its number of keys; the `len` of a list or tuple is its number of elements. For class instances, define the `__len__` method

and code the length calculation yourself, and then call `len(instance)` and Python will call your `__len__` special method for you.

4. `__delitem__` is called when you call `del instance[key]`, which you may remember as the way to delete individual items from a dictionary. When you use `del` on a class instance, Python calls the `__delitem__` special method for you.

In Java, you determine whether two string variables reference the same physical memory location by using `str1 == str2`. This is called *object identity*, and it is written in Python as `str1 is str2`. To compare string values in Java, you would use `str1.equals(str2)`; in Python, you would use `str1 == str2`. Java programmers who have been taught to believe that the world is a better place because `==` in Java compares by identity instead of by value may have a difficult time adjusting to Python's lack of such "gotchas".

At this point, you may be thinking, "All this work just to do something in a class that I can do with a built−in datatype." And it's true that life would be easier (and the entire `UserDict` class would be unnecessary) if you could inherit from built−in datatypes like a dictionary. But even if you could, special methods would still be useful, because they can be used in any class, not just wrapper classes like `UserDict`.

Special methods mean that *any class* can store key/value pairs like a dictionary, just by defining the `__setitem__` method. *Any class* can act like a sequence, just by defining the `__getitem__` method. Any class that defines the `__cmp__` method can be compared with `==`. And if your class represents something that has a length, don't define a `GetLength` method; define the `__len__` method and use `len(instance)`.

While other object−oriented languages only let you define the physical model of an object ("this object has a `GetLength` method"), Python's special class methods like `__len__` allow you to define the logical model of an object ("this object has a length").

Python has a lot of other special methods. There's a whole set of them that let classes act like numbers, allowing you to add, subtract, and do other arithmetic operations on class instances. (The canonical example of this is a class that represents complex numbers, numbers with both real and imaginary components.) The `__call__` method lets a class act like a function, allowing you to call a class instance directly. And there are other special methods that allow classes to have read−only and write−only data attributes; you'll talk more about those in later chapters.

Further Reading on Special Class Methods

- *Python Reference Manual* (http://www.python.org/doc/current/ref/) documents all the special class methods (http://www.python.org/doc/current/ref/specialnames.html).

5.8. Introducing Class Attributes

You already know about data attributes, which are variables owned by a specific instance of a class. Python also supports class attributes, which are variables owned by the class itself.

Example 5.17. Introducing Class Attributes

```
class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}
```

```
>>> import fileinfo
>>> fileinfo.MP3FileInfo                  ❶
<class fileinfo.MP3FileInfo at 01257FDC>
>>> fileinfo.MP3FileInfo.tagDataMap ❷
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built-in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
>>> m = fileinfo.MP3FileInfo()            ❸
>>> m.tagDataMap
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built-in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
```

❶ `MP3FileInfo` is the class itself, not any particular instance of the class.

❷ `tagDataMap` is a class attribute: literally, an attribute of the class. It is available before creating any instances of the class.

❸ Class attributes are available both through direct reference to the class and through any instance of the class.

In Java, both static variables (called class attributes in Python) and instance variables (called data attributes in Python) are defined immediately after the class definition (one with the `static` keyword, one without). In Python, only class attributes can be defined here; data attributes are defined in the `__init__` method.

Class attributes can be used as class−level constants (which is how you use them in `MP3FileInfo`), but they are not really constants. You can also change them.

There are no constants in Python. Everything can be changed if you try hard enough. This fits with one of the core principles of Python: bad behavior should be discouraged but not banned. If you really want to change the value of `None`, you can do it, but don't come running to me when your code is impossible to debug.

Example 5.18. Modifying Class Attributes

```
>>> class counter:
...     count = 0                     ❶
...     def __init__(self):
...         self.__class__.count += 1 ❷
...
>>> counter
<class __main__.counter at 010EAECC>
>>> counter.count                     ❸
0
>>> c = counter()
>>> c.count                           ❹
1
>>> counter.count
1
>>> d = counter()                     ❺
>>> d.count
2
>>> c.count
2
>>> counter.count
```

1. `count` is a class attribute of the `counter` class.
2. `__class__` is a built−in attribute of every class instance (of every class). It is a reference to the class that `self` is an instance of (in this case, the `counter` class).
3. Because `count` is a class attribute, it is available through direct reference to the class, before you have created any instances of the class.
4. Creating an instance of the class calls the `__init__` method, which increments the class attribute `count` by `1`. This affects the class itself, not just the newly created instance.
5. Creating a second instance will increment the class attribute `count` again. Notice how the class attribute is shared by the class and all instances of the class.

5.9. Private Functions

Like most languages, Python has the concept of private elements:

* Private functions, which can't be called from outside their module
* Private class methods, which can't be called from outside their class
* Private attributes, which can't be accessed from outside their class.

Unlike in most languages, whether a Python function, method, or attribute is private or public is determined entirely by its name.

If the name of a Python function, class method, or attribute starts with (but doesn't end with) two underscores, it's private; everything else is public. Python has no concept of *protected* class methods (accessible only in their own class and descendant classes). Class methods are either private (accessible only in their own class) or public (accessible from anywhere).

In `MP3FileInfo`, there are two methods: `__parse` and `__setitem__`. As you have already discussed, `__setitem__` is a special method; normally, you would call it indirectly by using the dictionary syntax on a class instance, but it is public, and you could call it directly (even from outside the `fileinfo` module) if you had a really good reason. However, `__parse` is private, because it has two underscores at the beginning of its name.

In Python, all special methods (like `__setitem__`) and built−in attributes (like `__doc__`) follow a standard naming convention: they both start with and end with two underscores. Don't name your own methods and attributes this way, because it will only confuse you (and others) later.

Example 5.19. Trying to Call a Private Method

```
>>> import fileinfo
>>> m = fileinfo.MP3FileInfo()
>>> m.__parse("/music/_singles/kairo.mp3") ❶
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'MP3FileInfo' instance has no attribute '__parse'
```

1. If you try to call a private method, Python will raise a slightly misleading exception, saying that the method does not exist. Of course it does exist, but it's private, so it's not accessible outside the class.Strictly speaking, private methods are accessible outside their class, just not *easily* accessible. Nothing in Python is truly private; internally, the names of private methods and attributes are mangled and unmangled on the fly to make them seem inaccessible by their given names. You can access the `__parse` method of the `MP3FileInfo` class by

the name `_MP3FileInfo__parse`. Acknowledge that this is interesting, but promise to never, ever do it in real code. Private methods are private for a reason, but like many other things in Python, their privateness is ultimately a matter of convention, not force.

Further Reading on Private Functions

* *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses the inner workings of private variables (http://www.python.org/doc/current/tut/node11.html#SECTION0011600000000000000000).

5.10. Summary

That's it for the hard−core object trickery. You'll see a real−world application of special class methods in Chapter 12, which uses `getattr` to create a proxy to a remote web service.

The next chapter will continue using this code sample to explore other Python concepts, such as exceptions, file objects, and `for` loops.

Before diving into the next chapter, make sure you're comfortable doing all of these things:

* Importing modules using either `import` *`module`* or `from` *`module`* `import`
* Defining and instantiating classes
* Defining `__init__` methods and other special class methods, and understanding when they are called
* Subclassing `UserDict` to define classes that act like dictionaries
* Defining data attributes and class attributes, and understanding the differences between them
* Defining private attributes and methods

Chapter 6. Exceptions and File Handling

In this chapter, you will dive into exceptions, file objects, `for` loops, and the `os` and `sys` modules. If you've used exceptions in another programming language, you can skim the first section to get a sense of Python's syntax. Be sure to tune in again for file handling.

6.1. Handling Exceptions

Like many other programming languages, Python has exception handling via `try...except` blocks.

Python uses `try...except` to handle exceptions and `raise` to generate them. Java and C++ use `try...catch` to handle exceptions, and `throw` to generate them.

Exceptions are everywhere in Python. Virtually every module in the standard Python library uses them, and Python itself will raise them in a lot of different circumstances. You've already seen them repeatedly throughout this book.

- Accessing a non−existent dictionary key will raise a `KeyError` exception.
- Searching a list for a non−existent value will raise a `ValueError` exception.
- Calling a non−existent method will raise an `AttributeError` exception.
- Referencing a non−existent variable will raise a `NameError` exception.
- Mixing datatypes without coercion will raise a `TypeError` exception.

In each of these cases, you were simply playing around in the Python IDE: an error occurred, the exception was printed (depending on your IDE, perhaps in an intentionally jarring shade of red), and that was that. This is called an *unhandled* exception. When the exception was raised, there was no code to explicitly notice it and deal with it, so it bubbled its way back to the default behavior built in to Python, which is to spit out some debugging information and give up. In the IDE, that's no big deal, but if that happened while your actual Python program was running, the entire program would come to a screeching halt.

An exception doesn't need result in a complete program crash, though. Exceptions, when raised, can be *handled*. Sometimes an exception is really because you have a bug in your code (like accessing a variable that doesn't exist), but many times, an exception is something you can anticipate. If you're opening a file, it might not exist. If you're connecting to a database, it might be unavailable, or you might not have the correct security credentials to access it. If you know a line of code may raise an exception, you should handle the exception using a `try...except` block.

Example 6.1. Opening a Non−Existent File

```
>>> fsock = open("/notthere", "r")       ❶
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
IOError: [Errno 2] No such file or directory: '/notthere'
>>> try:
...     fsock = open("/notthere")       ❷
... except IOError:                     ❸
...     print "The file does not exist, exiting gracefully"
... print "This line will always print" ❹
The file does not exist, exiting gracefully
This line will always print
```

❶ Using the built−in `open` function, you can try to open a file for reading (more on `open` in the next section). But the file doesn't exist, so this raises the `IOError` exception. Since you haven't provided any explicit check

for an `IOError` exception, Python just prints out some debugging information about what happened and then gives up.

❷ You're trying to open the same non−existent file, but this time you're doing it within a `try...except` block.

❸ When the `open` method raises an `IOError` exception, you're ready for it. The `except IOError:` line catches the exception and executes your own block of code, which in this case just prints a more pleasant error message.

❹ Once an exception has been handled, processing continues normally on the first line after the `try...except` block. Note that this line will always print, whether or not an exception occurs. If you really did have a file called `notthere` in your root directory, the call to `open` would succeed, the `except` clause would be ignored, and this line would still be executed.

Exceptions may seem unfriendly (after all, if you don't catch the exception, your entire program will crash), but consider the alternative. Would you rather get back an unusable file object to a non−existent file? You'd need to check its validity somehow anyway, and if you forgot, somewhere down the line, your program would give you strange errors somewhere down the line that you would need to trace back to the source. I'm sure you've experienced this, and you know it's not fun. With exceptions, errors occur immediately, and you can handle them in a standard way at the source of the problem.

6.1.1. Using Exceptions For Other Purposes

There are a lot of other uses for exceptions besides handling actual error conditions. A common use in the standard Python library is to try to import a module, and then check whether it worked. Importing a module that does not exist will raise an `ImportError` exception. You can use this to define multiple levels of functionality based on which modules are available at run−time, or to support multiple platforms (where platform−specific code is separated into different modules).

You can also define your own exceptions by creating a class that inherits from the built−in `Exception` class, and then raise your exceptions with the `raise` command. See the further reading section if you're interested in doing this.

The next example demonstrates how to use an exception to support platform−specific functionality. This code comes from the `getpass` module, a wrapper module for getting a password from the user. Getting a password is accomplished differently on UNIX, Windows, and Mac OS platforms, but this code encapsulates all of those differences.

Example 6.2. Supporting Platform−Specific Functionality

```
# Bind the name getpass to the appropriate function
try:
    import termios, TERMIOS                     ❶
except ImportError:
    try:
        import msvcrt                           ❷
    except ImportError:
        try:
            from EasyDialogs import AskPassword ❸
        except ImportError:
            getpass = default_getpass           ❹
        else:                                   ❺
            getpass = AskPassword
    else:
        getpass = win_getpass
else:
    getpass = unix_getpass
```

❶ `termios` is a UNIX−specific module that provides low−level control over the input terminal. If this module is not available (because it's not on your system, or your system doesn't support it), the import fails and Python raises an `ImportError`, which you catch.

❷ OK, you didn't have `termios`, so let's try `msvcrt`, which is a Windows−specific module that provides an API to many useful functions in the Microsoft Visual C++ runtime services. If this import fails, Python will raise an `ImportError`, which you catch.

❸ If the first two didn't work, you try to import a function from `EasyDialogs`, which is a Mac OS−specific module that provides functions to pop up dialog boxes of various types. Once again, if this import fails, Python will raise an `ImportError`, which you catch.

❹ None of these platform−specific modules is available (which is possible, since Python has been ported to a lot of different platforms), so you need to fall back on a default password input function (which is defined elsewhere in the `getpass` module). Notice what you're doing here: assigning the function `default_getpass` to the variable `getpass`. If you read the official `getpass` documentation, it tells you that the `getpass` module defines a `getpass` function. It does this by binding `getpass` to the correct function for your platform. Then when you call the `getpass` function, you're really calling a platform−specific function that this code has set up for you. You don't need to know or care which platform your code is running on −− just call `getpass`, and it will always do the right thing.

❺ A `try...except` block can have an `else` clause, like an `if` statement. If no exception is raised during the `try` block, the `else` clause is executed afterwards. In this case, that means that the `from EasyDialogs import AskPassword` import worked, so you should bind `getpass` to the `AskPassword` function. Each of the other `try...except` blocks has similar `else` clauses to bind `getpass` to the appropriate function when you find an `import` that works.

Further Reading on Exception Handling

- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses defining and raising your own exceptions, and handling multiple exceptions at once (http://www.python.org/doc/current/tut/node10.html#SECTION0010400000000000000000).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes all the built−in exceptions (http://www.python.org/doc/current/lib/module−exceptions.html).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the getpass (http://www.python.org/doc/current/lib/module−getpass.html) module.
- *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the `traceback` module (http://www.python.org/doc/current/lib/module−traceback.html), which provides low−level access to exception attributes after an exception is raised.
- *Python Reference Manual* (http://www.python.org/doc/current/ref/) discusses the inner workings of the `try...except` block (http://www.python.org/doc/current/ref/try.html).

6.2. Working with File Objects

Python has a built−in function, `open`, for opening a file on disk. `open` returns a file object, which has methods and attributes for getting information about and manipulating the opened file.

Example 6.3. Opening a File

```
>>> f = open("/music/_singles/kairo.mp3", "rb") ❶
>>> f                                           ❷
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.mode                                      ❸
'rb'
>>> f.name                                      ❹
'/music/_singles/kairo.mp3'
```

❶ The `open` method can take up to three parameters: a filename, a mode, and a buffering parameter. Only the first one, the filename, is required; the other two are optional. If not specified, the file is opened for reading in text mode. Here you are opening the file for reading in binary mode. (`print open.__doc__` displays a great explanation of all the possible modes.)

❷ The `open` function returns an object (by now, this should not surprise you). A file object has several useful attributes.

❸ The `mode` attribute of a file object tells you in which mode the file was opened.

❹ The `name` attribute of a file object tells you the name of the file that the file object has open.

6.2.1. Reading Files

After you open a file, the first thing you'll want to do is read from it, as shown in the next example.

Example 6.4. Reading a File

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.tell()              ❶
0
>>> f.seek(-128, 2)       ❷
>>> f.tell()              ❸
7542909
>>> tagData = f.read(128) ❹
>>> tagData
'TAGKAIRO****THE BEST GOA         ***DJ MARY-JANE***
Rave Mix                      2000http://mp3.com/DJMARYJANE     \037'
>>> f.tell()              ❺
7543037
```

❶ A file object maintains state about the file it has open. The `tell` method of a file object tells you your current position in the open file. Since you haven't done anything with this file yet, the current position is `0`, which is the beginning of the file.

❷ The `seek` method of a file object moves to another position in the open file. The second parameter specifies what the first one means; `0` means move to an absolute position (counting from the start of the file), `1` means move to a relative position (counting from the current position), and `2` means move to a position relative to the end of the file. Since the MP3 tags you're looking for are stored at the end of the file, you use `2` and tell the file object to move to a position `128` bytes from the end of the file.

❸ The `tell` method confirms that the current file position has moved.

❹ The `read` method reads a specified number of bytes from the open file and returns a string with the data that was read. The optional parameter specifies the maximum number of bytes to read. If no parameter is specified, `read` will read until the end of the file. (You could have simply said `read()` here, since you know exactly where you are in the file and you are, in fact, reading the last 128 bytes.) The read data is assigned to the `tagData` variable, and the current position is updated based on how many bytes were read.

❺ The `tell` method confirms that the current position has moved. If you do the math, you'll see that after reading 128 bytes, the position has been incremented by 128.

6.2.2. Closing Files

Open files consume system resources, and depending on the file mode, other programs may not be able to access them. It's important to close files as soon as you're finished with them.

Example 6.5. Closing a File

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed       ❶
False
>>> f.close()      ❷
>>> f
<closed file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed       ❸
True
>>> f.seek(0)      ❹
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.tell()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.read()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.close()      ❺
```

❶ The `closed` attribute of a file object indicates whether the object has a file open or not. In this case, the file is still open (`closed` is `False`).

❷ To close a file, call the `close` method of the file object. This frees the lock (if any) that you were holding on the file, flushes buffered writes (if any) that the system hadn't gotten around to actually writing yet, and releases the system resources.

❸ The `closed` attribute confirms that the file is closed.

❹ Just because a file is closed doesn't mean that the file object ceases to exist. The variable `f` will continue to exist until it goes out of scope or gets manually deleted. However, none of the methods that manipulate an open file will work once the file has been closed; they all raise an exception.

❺ Calling `close` on a file object whose file is already closed does *not* raise an exception; it fails silently.

6.2.3. Handling I/O Errors

Now you've seen enough to understand the file handling code in the `fileinfo.py` sample code from teh previous chapter. This example shows how to safely open and read from a file and gracefully handle errors.

Example 6.6. File Objects in `MP3FileInfo`

```
        try:                                ❶
            fsock = open(filename, "rb", 0) ❷
            try:
                fsock.seek(-128, 2)         ❸
                tagdata = fsock.read(128)   ❹
            finally:                        ❺
                fsock.close()
            .
            .
            .
        except IOError:                     ❻
            pass
```

❶ Because opening and reading files is risky and may raise an exception, all of this code is wrapped in a `try...except` block. (Hey, isn't standardized indentation great? This is where you start to appreciate it.)

❷ The `open` function may raise an `IOError`. (Maybe the file doesn't exist.)

❸ The `seek` method may raise an `IOError`. (Maybe the file is smaller than 128 bytes.)

❹ The `read` method may raise an `IOError`. (Maybe the disk has a bad sector, or it's on a network drive and the network just went down.)

❺ This is new: a `try...finally` block. Once the file has been opened successfully by the `open` function, you want to make absolutely sure that you close it, even if an exception is raised by the `seek` or `read` methods. That's what a `try...finally` block is for: code in the `finally` block will *always* be executed, even if something in the `try` block raises an exception. Think of it as code that gets executed on the way out, regardless of what happened before.

❻ At last, you handle your `IOError` exception. This could be the `IOError` exception raised by the call to `open`, `seek`, or `read`. Here, you really don't care, because all you're going to do is ignore it silently and continue. (Remember, `pass` is a Python statement that does nothing.) That's perfectly legal; "handling" an exception can mean explicitly doing nothing. It still counts as handled, and processing will continue normally on the next line of code after the `try...except` block.

6.2.4. Writing to Files

As you would expect, you can also write to files in much the same way that you read from them. There are two basic file modes:

- "Append" mode will add data to the end of the file.
- "write" mode will overwrite the file.

Either mode will create the file automatically if it doesn't already exist, so there's never a need for any sort of fiddly "if the log file doesn't exist yet, create a new empty file just so you can open it for the first time" logic. Just open it and start writing.

Example 6.7. Writing to Files

```
>>> logfile = open('test.log', 'w') ❶
>>> logfile.write('test succeeded') ❷
>>> logfile.close()
>>> print file('test.log').read()   ❸
test succeeded
>>> logfile = open('test.log', 'a') ❹
>>> logfile.write('line 2')
>>> logfile.close()
>>> print file('test.log').read()   ❺
test succeededline 2
```

❶ You start boldly by creating either the new file `test.log` or overwrites the existing file, and opening the file for writing. (The second parameter `"w"` means open the file for writing.) Yes, that's all as dangerous as it sounds. I hope you didn't care about the previous contents of that file, because it's gone now.

❷ You can add data to the newly opened file with the `write` method of the file object returned by `open`.

❸ `file` is a synonym for `open`. This one−liner opens the file, reads its contents, and prints them.

❹ You happen to know that `test.log` exists (since you just finished writing to it), so you can open it and append to it. (The `"a"` parameter means open the file for appending.) Actually you could do this even if the file didn't exist, because opening the file for appending will create the file if necessary. But appending

will *never* harm the existing contents of the file.

❺ As you can see, both the original line you wrote and the second line you appended are now in `test.log`. Also note that carriage returns are not included. Since you didn't write them explicitly to the file either time, the file doesn't include them. You can write a carriage return with the `"\n"` character. Since you didn't do this, everything you wrote to the file ended up smooshed together on the same line.

Further Reading on File Handling

- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses reading and writing files, including how to read a file one line at a time into a list (http://www.python.org/doc/current/tut/node9.html#SECTION009210000000000000000).
- eff−bot (http://www.effbot.org/guides/) discusses efficiency and performance of various ways of reading a file (http://www.effbot.org/guides/readline−performance.htm).
- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) answers common questions about files (http://www.faqts.com/knowledge−base/index.phtml/fid/552).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes all the file object methods (http://www.python.org/doc/current/lib/bltin−file−objects.html).

6.3. Iterating with `for` Loops

Like most other languages, Python has `for` loops. The only reason you haven't seen them until now is that Python is good at so many other things that you don't need them as often.

Most other languages don't have a powerful list datatype like Python, so you end up doing a lot of manual work, specifying a start, end, and step to define a range of integers or characters or other iteratable entities. But in Python, a `for` loop simply iterates over a list, the same way list comprehensions work.

Example 6.8. Introducing the `for` Loop

```
>>> li = ['a', 'b', 'e']
>>> for s in li:          ❶
...     print s           ❷
a
b
e
>>> print "\n".join(li)  ❸
a
b
e
```

❶ The syntax for a `for` loop is similar to list comprehensions. `li` is a list, and `s` will take the value of each element in turn, starting from the first element.

❷ Like an `if` statement or any other indented block, a `for` loop can have any number of lines of code in it.

❸ This is the reason you haven't seen the `for` loop yet: you haven't needed it yet. It's amazing how often you use `for` loops in other languages when all you really want is a `join` or a list comprehension.

Doing a "normal" (by Visual Basic standards) counter `for` loop is also simple.

Example 6.9. Simple Counters

```
>>> for i in range(5):             ❶
...     print i
```

```
0
1
2
3
4
>>> li = ['a', 'b', 'c', 'd', 'e']
>>> for i in range(len(li)):        ❷
...     print li[i]
a
b
c
d
e
```

❶ As you saw in Example 3.20, Assigning Consecutive Values , `range` produces a list of integers, which you then loop through. I know it looks a bit odd, but it is occasionally (and I stress *occasionally*) useful to have a counter loop.

❷ Don't ever do this. This is Visual Basic−style thinking. Break out of it. Just iterate through the list, as shown in the previous example.

`for` loops are not just for simple counters. They can iterate through all kinds of things. Here is an example of using a `for` loop to iterate through a dictionary.

Example 6.10. Iterating Through a Dictionary

```
>>> import os
>>> for k, v in os.environ.items():      ❶ ❷
...     print "%s=%s" % (k, v)
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim

[...snip...]
>>> print "\n".join(["%s=%s" % (k, v)
...     for k, v in os.environ.items()]) ❸
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim

[...snip...]
```

❶ `os.environ` is a dictionary of the environment variables defined on your system. In Windows, these are your user and system variables accessible from MS−DOS. In UNIX, they are the variables exported in your shell's startup scripts. In Mac OS, there is no concept of environment variables, so this dictionary is empty.

❷ `os.environ.items()` returns a list of tuples: `[(key1, value1), (key2, value2), ...]`. The `for` loop iterates through this list. The first round, it assigns `key1` to `k` and `value1` to `v`, so `k` = `USERPROFILE` and `v` = `C:\Documents and Settings\mpilgrim`. In the second round, `k` gets the second key, `OS`, and `v` gets the corresponding value, `Windows_NT`.

❸ With multi−variable assignment and list comprehensions, you can replace the entire `for` loop with a single statement. Whether you actually do this in real code is a matter of personal coding style. I like it because it makes it clear that what I'm doing is mapping a dictionary into a list, then joining the list into a single string. Other programmers prefer to write this out as a `for` loop. The output is the same in either case, although this version is slightly faster, because there is only one `print` statement instead of many.

Now we can look at the `for` loop in `MP3FileInfo`, from the sample `fileinfo.py` program introduced in

Example 6.11. `for` Loop in `MP3FileInfo`

```
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}                        ❶
    .
    .
    .
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items(): ❷
                    self[tag] = parseFunc(tagdata[start:end])          ❸
```

❶ `tagDataMap` is a class attribute that defines the tags you're looking for in an MP3 file. Tags are stored in fixed−length fields. Once you read the last 128 bytes of the file, bytes 3 through 32 of those are always the song title, 33 through 62 are always the artist name, 63 through 92 are the album name, and so forth. Note that `tagDataMap` is a dictionary of tuples, and each tuple contains two integers and a function reference.

❷ This looks complicated, but it's not. The structure of the `for` variables matches the structure of the elements of the list returned by `items`. Remember that `items` returns a list of tuples of the form `(`*`key`*`, `*`value`*`)`. The first element of that list is `("title", (3, 33, <function stripnulls>))`, so the first time around the loop, `tag` gets `"title"`, `start` gets `3`, `end` gets `33`, and `parseFunc` gets the function `stripnulls`.

❸ Now that you've extracted all the parameters for a single MP3 tag, saving the tag data is easy. You slice `tagdata` from `start` to `end` to get the actual data for this tag, call `parseFunc` to post−process the data, and assign this as the value for the key `tag` in the pseudo−dictionary `self`. After iterating through all the elements in `tagDataMap`, `self` has the values for all the tags, and you know what that looks like.

6.4. Using `sys.modules`

Modules, like everything else in Python, are objects. Once imported, you can always get a reference to a module through the global dictionary `sys.modules`.

Example 6.12. Introducing `sys.modules`

```
>>> import sys                          ❶
>>> print '\n'.join(sys.modules.keys()) ❷
win32api
os.path
os
exceptions
__main__
ntpath
nt
sys
__builtin__
site
signal
UserDict
stat
```

❶ The `sys` module contains system−level information, such as the version of Python you're

running (`sys.version` or `sys.version_info`), and system−level options such as the maximum allowed recursion depth (`sys.getrecursionlimit()` and `sys.setrecursionlimit()`).

❷ `sys.modules` is a dictionary containing all the modules that have ever been imported since Python was started; the key is the module name, the value is the module object. Note that this is more than just the modules *your* program has imported. Python preloads some modules on startup, and if you're using a Python IDE, `sys.modules` contains all the modules imported by all the programs you've run within the IDE.

This example demonstrates how to use `sys.modules`.

Example 6.13. Using `sys.modules`

```
>>> import fileinfo         ❶
>>> print '\n'.join(sys.modules.keys())
win32api
os.path
os
fileinfo
exceptions
__main__
ntpath
nt
sys
__builtin__
site
signal
UserDict
stat
>>> fileinfo
<module 'fileinfo' from 'fileinfo.pyc'>
>>> sys.modules["fileinfo"] ❷
<module 'fileinfo' from 'fileinfo.pyc'>
```

❶ As new modules are imported, they are added to `sys.modules`. This explains why importing the same module twice is very fast: Python has already loaded and cached the module in `sys.modules`, so importing the second time is simply a dictionary lookup.

❷ Given the name (as a string) of any previously−imported module, you can get a reference to the module itself through the `sys.modules` dictionary.

The next example shows how to use the `__module__` class attribute with the `sys.modules` dictionary to get a reference to the module in which a class is defined.

Example 6.14. The `__module__` Class Attribute

```
>>> from fileinfo import MP3FileInfo
>>> MP3FileInfo.__module__              ❶
'fileinfo'
>>> sys.modules[MP3FileInfo.__module__] ❷
<module 'fileinfo' from 'fileinfo.pyc'>
```

❶ Every Python class has a built−in class attribute `__module__`, which is the name of the module in which the class is defined.

❷ Combining this with the `sys.modules` dictionary, you can get a reference to the module in which a class is defined.

Now you're ready to see how `sys.modules` is used in `fileinfo.py`, the sample program introduced in Chapter 5. This example shows that portion of the code.

Example 6.15. `sys.modules` in `fileinfo.py`

```
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):       ❶
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]        ❷
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo ❸
```

❶ This is a function with two arguments; `filename` is required, but `module` is optional and defaults to the module that contains the `FileInfo` class. This looks inefficient, because you might expect Python to evaluate the `sys.modules` expression every time the function is called. In fact, Python evaluates default expressions only once, the first time the module is imported. As you'll see later, you never call this function with a `module` argument, so `module` serves as a function−level constant.

❷ You'll plow through this line later, after you dive into the `os` module. For now, take it on faith that `subclass` ends up as the name of a class, like `MP3FileInfo`.

❸ You already know about `getattr`, which gets a reference to an object by name. `hasattr` is a complementary function that checks whether an object has a particular attribute; in this case, whether a module has a particular class (although it works for any object and any attribute, just like `getattr`). In English, this line of code says, "If this module has the class named by `subclass` then return it, otherwise return the base class `FileInfo`."

Further Reading on Modules

- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses exactly when and how default arguments are evaluated (http://www.python.org/doc/current/tut/node6.html#SECTION006710000000000000000).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the `sys` (http://www.python.org/doc/current/lib/module−sys.html) module.

6.5. Working with Directories

The `os.path` module has several functions for manipulating files and directories. Here, we're looking at handling pathnames and listing the contents of a directory.

Example 6.16. Constructing Pathnames

```
>>> import os
>>> os.path.join("c:\\music\\ap\\", "mahadeva.mp3") ❶ ❷
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.join("c:\\music\\ap", "mahadeva.mp3")   ❸
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.expanduser("~")                         ❹
'c:\\Documents and Settings\\mpilgrim\\My Documents'
>>> os.path.join(os.path.expanduser("~"), "Python") ❺
'c:\\Documents and Settings\\mpilgrim\\My Documents\\Python'
```

❶ `os.path` is a reference to a module −− which module depends on your platform. Just as `getpass` encapsulates differences between platforms by setting `getpass` to a platform−specific function, `os` encapsulates differences between platforms by setting `path` to a platform−specific module.

❷

The `join` function of `os.path` constructs a pathname out of one or more partial pathnames. In this case, it simply concatenates strings. (Note that dealing with pathnames on Windows is annoying because the backslash character must be escaped.)

❸ In this slightly less trivial case, `join` will add an extra backslash to the pathname before joining it to the filename. I was overjoyed when I discovered this, since `addSlashIfNecessary` is one of the stupid little functions I always need to write when building up my toolbox in a new language. *Do not* write this stupid little function in Python; smart people have already taken care of it for you.

❹ `expanduser` will expand a pathname that uses ~ to represent the current user's home directory. This works on any platform where users have a home directory, like Windows, UNIX, and Mac OS X; it has no effect on Mac OS.

❺ Combining these techniques, you can easily construct pathnames for directories and files under the user's home directory.

Example 6.17. Splitting Pathnames

```
>>> os.path.split("c:\\music\\ap\\mahadeva.mp3")                        ❶
('c:\\music\\ap', 'mahadeva.mp3')
>>> (filepath, filename) = os.path.split("c:\\music\\ap\\mahadeva.mp3") ❷
>>> filepath                                                            ❸
'c:\\music\\ap'
>>> filename                                                            ❹
'mahadeva.mp3'
>>> (shortname, extension) = os.path.splitext(filename)                 ❺
>>> shortname
'mahadeva'
>>> extension
'.mp3'
```

❶ The `split` function splits a full pathname and returns a tuple containing the path and filename. Remember when I said you could use multi−variable assignment to return multiple values from a function? Well, `split` is such a function.

❷ You assign the return value of the `split` function into a tuple of two variables. Each variable receives the value of the corresponding element of the returned tuple.

❸ The first variable, `filepath`, receives the value of the first element of the tuple returned from `split`, the file path.

❹ The second variable, `filename`, receives the value of the second element of the tuple returned from `split`, the filename.

❺ `os.path` also contains a function `splitext`, which splits a filename and returns a tuple containing the filename and the file extension. You use the same technique to assign each of them to separate variables.

Example 6.18. Listing Directories

```
>>> os.listdir("c:\\music\\_singles\\")                 ❶
['a_time_long_forgotten_con.mp3', 'hellraiser.mp3',
'kairo.mp3', 'long_way_home1.mp3', 'sidewinder.mp3',
'spinning.mp3']
>>> dirname = "c:\\"
>>> os.listdir(dirname)                                 ❷
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'cygwin',
'docbook', 'Documents and Settings', 'Incoming', 'Inetpub', 'IO.SYS',
'MSDOS.SYS', 'Music', 'NTDETECT.COM', 'ntldr', 'pagefile.sys',
'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
>>> [f for f in os.listdir(dirname)
...     if os.path.isfile(os.path.join(dirname, f))]    ❸
```

```
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'IO.SYS', 'MSDOS.SYS',
'NTDETECT.COM', 'ntldr', 'pagefile.sys']
>>> [f for f in os.listdir(dirname)
...     if os.path.isdir(os.path.join(dirname, f))]  ❹
['cygwin', 'docbook', 'Documents and Settings', 'Incoming',
'Inetpub', 'Music', 'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
```

❶ The `listdir` function takes a pathname and returns a list of the contents of the directory.

❷ `listdir` returns both files and folders, with no indication of which is which.

❸ You can use list filtering and the `isfile` function of the `os.path` module to separate the files from the folders. `isfile` takes a pathname and returns 1 if the path represents a file, and 0 otherwise. Here you're using `os.path.join` to ensure a full pathname, but `isfile` also works with a partial path, relative to the current working directory. You can use `os.getcwd()` to get the current working directory.

❹ `os.path` also has a `isdir` function which returns 1 if the path represents a directory, and 0 otherwise. You can use this to get a list of the subdirectories within a directory.

Example 6.19. Listing Directories in `fileinfo.py`

```
def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]            ❶ ❷
    fileList = [os.path.join(directory, f)
               for f in fileList
                if os.path.splitext(f)[1] in fileExtList]  ❸ ❹ ❺
```

❶ `os.listdir(directory)` returns a list of all the files and folders in `directory`.

❷ Iterating through the list with `f`, you use `os.path.normcase(f)` to normalize the case according to operating system defaults. `normcase` is a useful little function that compensates for case−insensitive operating systems that think that `mahadeva.mp3` and `mahadeva.MP3` are the same file. For instance, on Windows and Mac OS, `normcase` will convert the entire filename to lowercase; on UNIX−compatible systems, it will return the filename unchanged.

❸ Iterating through the normalized list with `f` again, you use `os.path.splitext(f)` to split each filename into name and extension.

❹ For each file, you see if the extension is in the list of file extensions you care about (`fileExtList`, which was passed to the `listDirectory` function).

❺ For each file you care about, you use `os.path.join(directory, f)` to construct the full pathname of the file, and return a list of the full pathnames.

Whenever possible, you should use the functions in `os` and `os.path` for file, directory, and path manipulations. These modules are wrappers for platform−specific modules, so functions like `os.path.split` work on UNIX, Windows, Mac OS, and any other platform supported by Python.

There is one other way to get the contents of a directory. It's very powerful, and it uses the sort of wildcards that you may already be familiar with from working on the command line.

Example 6.20. Listing Directories with `glob`

```
>>> os.listdir("c:\\music\\_singles\\")              ❶
['a_time_long_forgotten_con.mp3', 'hellraiser.mp3',
'kairo.mp3', 'long_way_home1.mp3', 'sidewinder.mp3',
'spinning.mp3']
```

```
>>> import glob
>>> glob.glob('c:\\music\\_singles\\*.mp3')                ❷
['c:\\music\\_singles\\a_time_long_forgotten_con.mp3',
'c:\\music\\_singles\\hellraiser.mp3',
'c:\\music\\_singles\\kairo.mp3',
'c:\\music\\_singles\\long_way_home1.mp3',
'c:\\music\\_singles\\sidewinder.mp3',
'c:\\music\\_singles\\spinning.mp3']
>>> glob.glob('c:\\music\\_singles\\s*.mp3')               ❸
['c:\\music\\_singles\\sidewinder.mp3',
'c:\\music\\_singles\\spinning.mp3']
>>> glob.glob('c:\\music\\*\\*.mp3')                       ❹
```

❶ As you saw earlier, `os.listdir` simply takes a directory path and lists all files and directories in that directory.

❷ The `glob` module, on the other hand, takes a wildcard and returns the full path of all files and directories matching the wildcard. Here the wildcard is a directory path plus "*.mp3", which will match all `.mp3` files. Note that each element of the returned list already includes the full path of the file.

❸ If you want to find all the files in a specific directory that start with "s" and end with ".mp3", you can do that too.

❹ Now consider this scenario: you have a `music` directory, with several subdirectories within it, with `.mp3` files within each subdirectory. You can get a list of all of those with a single call to `glob`, by using two wildcards at once. One wildcard is the `"*.mp3"` (to match `.mp3` files), and one wildcard is *within the directory path itself*, to match any subdirectory within `c:\music`. That's a crazy amount of power packed into one deceptively simple−looking function!

Further Reading on the `os` Module

- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) answers questions about the `os` module (http://www.faqts.com/knowledge−base/index.phtml/fid/240).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the `os` (http://www.python.org/doc/current/lib/module−os.html) module and the `os.path` (http://www.python.org/doc/current/lib/module−os.path.html) module.

6.6. Putting It All Together

Once again, all the dominoes are in place. You've seen how each line of code works. Now let's step back and see how it all fits together.

Example 6.21. `listDirectory`

```
def listDirectory(directory, fileExtList):                                          ❶
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f)
               for f in fileList
                if os.path.splitext(f)[1] in fileExtList]                           ❷
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):        ❸
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]         ❹
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo  ❺
    return [getFileInfoClass(f)(f) for f in fileList]                               ❻
```

❶

`listDirectory` is the main attraction of this entire module. It takes a directory (like `c:\music\_singles\` in my case) and a list of interesting file extensions (like `['.mp3']`), and it returns a list of class instances that act like dictionaries that contain metadata about each interesting file in that directory. And it does it in just a few straightforward lines of code.

❷ As you saw in the previous section, this line of code gets a list of the full pathnames of all the files in `directory` that have an interesting file extension (as specified by `fileExtList`).

❸ Old−school Pascal programmers may be familiar with them, but most people give me a blank stare when I tell them that Python supports *nested functions* −− literally, a function within a function. The nested function `getFileInfoClass` can be called only from the function in which it is defined, `listDirectory`. As with any other function, you don't need an interface declaration or anything fancy; just define the function and code it.

❹ Now that you've seen the `os` module, this line should make more sense. It gets the extension of the file (`os.path.splitext(filename)[1]`), forces it to uppercase (`.upper()`), slices off the dot (`[1:]`), and constructs a class name out of it with string formatting. So `c:\music\ap\mahadeva.mp3` becomes `.mp3` becomes `.MP3` becomes `MP3` becomes `MP3FileInfo`.

❺ Having constructed the name of the handler class that would handle this file, you check to see if that handler class actually exists in this module. If it does, you return the class, otherwise you return the base class `FileInfo`. This is a very important point: *this function returns a class*. Not an instance of a class, but the class itself.

❻ For each file in the "interesting files" list (`fileList`), you call `getFileInfoClass` with the filename (`f`). Calling `getFileInfoClass(f)` returns a class; you don't know exactly which class, but you don't care. You then create an instance of this class (whatever it is) and pass the filename (`f` again), to the `__init__` method. As you saw earlier in this chapter, the `__init__` method of `FileInfo` sets `self["name"]`, which triggers `__setitem__`, which is overridden in the descendant (`MP3FileInfo`) to parse the file appropriately to pull out the file's metadata. You do all that for each interesting file and return a list of the resulting instances.

Note that `listDirectory` is completely generic. It doesn't know ahead of time which types of files it will be getting, or which classes are defined that could potentially handle those files. It inspects the directory for the files to process, and then introspects its own module to see what special handler classes (like `MP3FileInfo`) are defined. You can extend this program to handle other types of files simply by defining an appropriately−named class: `HTMLFileInfo` for HTML files, `DOCFileInfo` for Word `.doc` files, and so forth. `listDirectory` will handle them all, without modification, by handing off the real work to the appropriate classes and collating the results.

6.7. Summary

The `fileinfo.py` program introduced in Chapter 5 should now make perfect sense.

```
"""Framework for getting filetype−specific metadata.

Instantiate appropriate class with filename.  Returned object acts like a
dictionary, with key−value pairs for each piece of metadata.
    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])

Or use listDirectory function to get info on all files in a directory.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...

Framework can be extended by adding classes for particular file types, e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Each class is completely responsible for
parsing its files appropriately; see MP3FileInfo for example.
"""
import os
```

```
import sys
from UserDict import UserDict

def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()

class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename

class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}

    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
        self.clear()
        try:
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(−128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass

    def __setitem__(self, key, item):
        if key == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f)
               for f in fileList
                if os.path.splitext(f)[1] in fileExtList]
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]

if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]):
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
        print
```

Before diving into the next chapter, make sure you're comfortable doing the following things:

* Catching exceptions with `try...except`
* Protecting external resources with `try...finally`
* Reading from files
* Assigning multiple values at once in a `for` loop
* Using the `os` module for all your cross−platform file manipulation needs
* Dynamically instantiating classes of unknown type by treating classes as objects and passing them around

Chapter 7. Regular Expressions

Regular expressions are a powerful and standardized way of searching, replacing, and parsing text with complex patterns of characters. If you've used regular expressions in other languages (like Perl), the syntax will be very familiar, and you get by just reading the summary of the `re` module (http://www.python.org/doc/current/lib/module−re.html) to get an overview of the available functions and their arguments.

7.1. Diving In

Strings have methods for searching (`index`, `find`, and `count`), replacing (`replace`), and parsing (`split`), but they are limited to the simplest of cases. The search methods look for a single, hard−coded substring, and they are always case−sensitive. To do case−insensitive searches of a string `s`, you must call `s.lower()` or `s.upper()` and make sure your search strings are the appropriate case to match. The `replace` and `split` methods have the same limitations.

If what you're trying to do can be accomplished with string functions, you should use them. They're fast and simple and easy to read, and there's a lot to be said for fast, simple, readable code. But if you find yourself using a lot of different string functions with `if` statements to handle special cases, or if you're combining them with `split` and `join` and list comprehensions in weird unreadable ways, you may need to move up to regular expressions.

Although the regular expression syntax is tight and unlike normal code, the result can end up being *more* readable than a hand−rolled solution that uses a long chain of string functions. There are even ways of embedding comments within regular expressions to make them practically self−documenting.

7.2. Case Study: Street Addresses

This series of examples was inspired by a real−life problem I had in my day job several years ago, when I needed to scrub and standardize street addresses exported from a legacy system before importing them into a newer system. (See, I don't just make this stuff up; it's actually useful.) This example shows how I approached the problem.

Example 7.1. Matching at the End of a String

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.')                 ❶
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.')                 ❷
'100 NORTH BRD. RD.'
>>> s[:−4] + s[−4:].replace('ROAD', 'RD.') ❸
'100 NORTH BROAD RD.'
>>> import re                                ❹
>>> re.sub('ROAD$', 'RD.', s)                ❺ ❻
'100 NORTH BROAD RD.'
```

❶ My goal is to standardize a street address so that `'ROAD'` is always abbreviated as `'RD.'`. At first glance, I thought this was simple enough that I could just use the string method `replace`. After all, all the data was already uppercase, so case mismatches would not be a problem. And the search string, `'ROAD'`, was a constant. And in this deceptively simple example, `s.replace` does indeed work.

❷ Life, unfortunately, is full of counterexamples, and I quickly discovered this one. The problem here is that `'ROAD'` appears twice in the address, once as part of the street name `'BROAD'` and once as its own word. The

`replace` method sees these two occurrences and blindly replaces both of them; meanwhile, I see my addresses getting destroyed.

❸ To solve the problem of addresses with more than one `'ROAD'` substring, you could resort to something like this: only search and replace `'ROAD'` in the last four characters of the address (`s[-4:]`), and leave the string alone (`s[:-4]`). But you can see that this is already getting unwieldy. For example, the pattern is dependent on the length of the string you're replacing (if you were replacing `'STREET'` with `'ST.'`, you would need to use `s[:-6]` and `s[-6:].replace(...)`). Would you like to come back in six months and debug this? I know I wouldn't.

❹ It's time to move up to regular expressions. In Python, all functionality related to regular expressions is contained in the `re` module.

❺ Take a look at the first parameter: `'ROAD$'`. This is a simple regular expression that matches `'ROAD'` only when it occurs at the end of a string. The `$` means "end of the string". (There is a corresponding character, the caret `^`, which means "beginning of the string".)

❻ Using the `re.sub` function, you search the string `s` for the regular expression `'ROAD$'` and replace it with `'RD.'`. This matches the `ROAD` at the end of the string `s`, but does *not* match the `ROAD` that's part of the word `BROAD`, because that's in the middle of `s`.

Continuing with my story of scrubbing addresses, I soon discovered that the previous example, matching `'ROAD'` at the end of the address, was not good enough, because not all addresses included a street designation at all; some just ended with the street name. Most of the time, I got away with it, but if the street name was `'BROAD'`, then the regular expression would match `'ROAD'` at the end of the string as part of the word `'BROAD'`, which is not what I wanted.

Example 7.2. Matching Whole Words

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s)  ❶
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s)  ❷
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s)  ❸
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s) ❹
'100 BROAD RD. APT 3'
```

❶ What I *really* wanted was to match `'ROAD'` when it was at the end of the string *and* it was its own whole word, not a part of some larger word. To express this in a regular expression, you use `\b`, which means "a word boundary must occur right here". In Python, this is complicated by the fact that the `'\'` character in a string must itself be escaped. This is sometimes referred to as the backslash plague, and it is one reason why regular expressions are easier in Perl than in Python. On the down side, Perl mixes regular expressions with other syntax, so if you have a bug, it may be hard to tell whether it's a bug in syntax or a bug in your regular expression.

❷ To work around the backslash plague, you can use what is called a raw string, by prefixing the string with the letter `r`. This tells Python that nothing in this string should be escaped; `'\t'` is a tab character, but `r'\t'` is really the backslash character `\` followed by the letter `t`. I recommend always using raw strings when dealing with regular expressions; otherwise, things get too confusing too quickly (and regular expressions get confusing quickly enough all by themselves).

❸ *sigh* Unfortunately, I soon found more cases that contradicted my logic. In this case, the street address contained the word `'ROAD'` as a whole word by itself, but it wasn't at the end, because the address had an apartment number after the street designation. Because `'ROAD'` isn't at the very end of

the string, it doesn't match, so the entire call to re.sub ends up replacing nothing at all, and you get the original string back, which is not what you want.

4. To solve this problem, I removed the $ character and added another \b. Now the regular expression reads "match 'ROAD' when it's a whole word by itself anywhere in the string," whether at the end, the beginning, or somewhere in the middle.

7.3. Case Study: Roman Numerals

You've most likely seen Roman numerals, even if you didn't recognize them. You may have seen them in copyrights of old movies and television shows ("Copyright MCMXLVI" instead of "Copyright 1946"), or on the dedication walls of libraries or universities ("established MDCCCLXXXVIII" instead of "established 1888"). You may also have seen them in outlines and bibliographical references. It's a system of representing numbers that really does date back to the ancient Roman empire (hence the name).

In Roman numerals, there are seven characters that are repeated and combined in various ways to represent numbers.

- I = 1
- V = 5
- X = 10
- L = 50
- C = 100
- D = 500
- M = 1000

The following are some general rules for constructing Roman numerals:

- Characters are additive. I is 1, II is 2, and III is 3. VI is 6 (literally, "5 and 1"), VII is 7, and VIII is 8.
- The tens characters (I, X, C, and M) can be repeated up to three times. At 4, you need to subtract from the next highest fives character. You can't represent 4 as IIII; instead, it is represented as IV ("1 less than 5"). The number 40 is written as XL (10 less than 50), 41 as XLI, 42 as XLII, 43 as XLIII, and then 44 as XLIV (10 less than 50, then 1 less than 5).
- Similarly, at 9, you need to subtract from the next highest tens character: 8 is VIII, but 9 is IX (1 less than 10), not VIIII (since the I character can not be repeated four times). The number 90 is XC, 900 is CM.
- The fives characters can not be repeated. The number 10 is always represented as X, never as VV. The number 100 is always C, never LL.
- Roman numerals are always written highest to lowest, and read left to right, so the order the of characters matters very much. DC is 600; CD is a completely different number (400, 100 less than 500). CI is 101; IC is not even a valid Roman numeral (because you can't subtract 1 directly from 100; you would need to write it as XCIX, for 10 less than 100, then 1 less than 10).

7.3.1. Checking for Thousands

What would it take to validate that an arbitrary string is a valid Roman numeral? Let's take it one digit at a time. Since Roman numerals are always written highest to lowest, let's start with the highest: the thousands place. For numbers 1000 and higher, the thousands are represented by a series of M characters.

Example 7.3. Checking for Thousands

```
>>> import re
>>> pattern = '^M?M?M?$'        (1)
>>> re.search(pattern, 'M')     (2)
<SRE_Match object at 0106FB58>
```

```
>>> re.search(pattern, 'MM')    ❸
<SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM')   ❹
<SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM')  ❺
>>> re.search(pattern, '')      ❻
<SRE_Match object at 0106F4A8>
```

❶ This pattern has three parts:

- `^` to match what follows only at the beginning of the string. If this were not specified, the pattern would match no matter where the `M` characters were, which is not what you want. You want to make sure that the `M` characters, if they're there, are at the beginning of the string.
- `M?` to optionally match a single `M` character. Since this is repeated three times, you're matching anywhere from zero to three `M` characters in a row.
- `$` to match what precedes only at the end of the string. When combined with the `^` character at the beginning, this means that the pattern must match the entire string, with no other characters before or after the `M` characters.

❷ The essence of the `re` module is the `search` function, that takes a regular expression (`pattern`) and a string (`'M'`) to try to match against the regular expression. If a match is found, `search` returns an object which has various methods to describe the match; if no match is found, `search` returns `None`, the Python null value. All you care about at the moment is whether the pattern matches, which you can tell by just looking at the return value of `search`. `'M'` matches this regular expression, because the first optional `M` matches and the second and third optional `M` characters are ignored.

❸ `'MM'` matches because the first and second optional `M` characters match and the third `M` is ignored.

❹ `'MMM'` matches because all three `M` characters match.

❺ `'MMMM'` does not match. All three `M` characters match, but then the regular expression insists on the string ending (because of the `$` character), and the string doesn't end yet (because of the fourth `M`). So `search` returns `None`.

❻ Interestingly, an empty string also matches this regular expression, since all the `M` characters are optional.

7.3.2. Checking for Hundreds

The hundreds place is more difficult than the thousands, because there are several mutually exclusive ways it could be expressed, depending on its value.

- `100` = `C`
- `200` = `CC`
- `300` = `CCC`
- `400` = `CD`
- `500` = `D`
- `600` = `DC`
- `700` = `DCC`
- `800` = `DCCC`
- `900` = `CM`

So there are four possible patterns:

- `CM`
- `CD`
- Zero to three `C` characters (zero if the hundreds place is 0)
- `D`, followed by zero to three `C` characters

The last two patterns can be combined:

- an optional D, followed by zero to three C characters

This example shows how to validate the hundreds place of a Roman numeral.

Example 7.4. Checking for Hundreds

```
>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$' ❶
>>> re.search(pattern, 'MCM')            ❷
<SRE_Match object at 01070390>
>>> re.search(pattern, 'MD')             ❸
<SRE_Match object at 01073A50>
>>> re.search(pattern, 'MMMCCC')         ❹
<SRE_Match object at 010748A8>
>>> re.search(pattern, 'MCMC')           ❺
>>> re.search(pattern, '')               ❻
<SRE_Match object at 01071D98>
```

❶ This pattern starts out the same as the previous one, checking for the beginning of the string (^), then the thousands place (M?M?M?). Then it has the new part, in parentheses, which defines a set of three mutually exclusive patterns, separated by vertical bars: CM, CD, and D?C?C?C? (which is an optional D followed by zero to three optional C characters). The regular expression parser checks for each of these patterns in order (from left to right), takes the first one that matches, and ignores the rest.

❷ 'MCM' matches because the first M matches, the second and third M characters are ignored, and the CM matches (so the CD and D?C?C?C? patterns are never even considered). MCM is the Roman numeral representation of 1900.

❸ 'MD' matches because the first M matches, the second and third M characters are ignored, and the D?C?C?C? pattern matches D (each of the three C characters are optional and are ignored). MD is the Roman numeral representation of 1500.

❹ 'MMMCCC' matches because all three M characters match, and the D?C?C?C? pattern matches CCC (the D is optional and is ignored). MMMCCC is the Roman numeral representation of 3300.

❺ 'MCMC' does not match. The first M matches, the second and third M characters are ignored, and the CM matches, but then the $ does not match because you're not at the end of the string yet (you still have an unmatched C character). The C does *not* match as part of the D?C?C?C? pattern, because the mutually exclusive CM pattern has already matched.

❻ Interestingly, an empty string still matches this pattern, because all the M characters are optional and ignored, and the empty string matches the D?C?C?C? pattern where all the characters are optional and ignored.

Whew! See how quickly regular expressions can get nasty? And you've only covered the thousands and hundreds places of Roman numerals. But if you followed all that, the tens and ones places are easy, because they're exactly the same pattern. But let's look at another way to express the pattern.

7.4. Using the {n,m} Syntax

In the previous section, you were dealing with a pattern where the same character could be repeated up to three times. There is another way to express this in regular expressions, which some people find more readable. First look at the method we already used in the previous example.

Example 7.5. The Old Way: Every Character Optional

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')     ❶
<_sre.SRE_Match object at 0x008EE090>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MM')    ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MMM')  ❸
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMMM') ❹
>>>
```

❶ This matches the start of the string, and then the first optional `M`, but not the second and third `M` (but that's okay because they're optional), and then the end of the string.

❷ This matches the start of the string, and then the first and second optional `M`, but not the third `M` (but that's okay because it's optional), and then the end of the string.

❸ This matches the start of the string, and then all three optional `M`, and then the end of the string.

❹ This matches the start of the string, and then all three optional `M`, but then does not match the the end of the string (because there is still one unmatched `M`), so the pattern does not match and returns `None`.

Example 7.6. The New Way: From `n` o `m`

```
>>> pattern = '^M{0,3}$'       ❶
>>> re.search(pattern, 'M')    ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MM')    ❸
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMM')  ❹
<_sre.SRE_Match object at 0x008EEDA8>
>>> re.search(pattern, 'MMMM') ❺
>>>
```

❶ This pattern says: "Match the start of the string, then anywhere from zero to three `M` characters, then the end of the string." The 0 and 3 can be any numbers; if you want to match at least one but no more than three `M` characters, you could say `M{1,3}`.

❷ This matches the start of the string, then one `M` out of a possible three, then the end of the string.

❸ This matches the start of the string, then two `M` out of a possible three, then the end of the string.

❹ This matches the start of the string, then three `M` out of a possible three, then the end of the string.

❺ This matches the start of the string, then three `M` out of a possible three, but then *does not match* the end of the string. The regular expression allows for up to only three `M` characters before the end of the string, but you have four, so the pattern does not match and returns `None`.

There is no way to programmatically determine that two regular expressions are equivalent. The best you can do is write a lot of test cases to make sure they behave the same way on all relevant inputs. You'll talk more about writing test cases later in this book.

7.4.1. Checking for Tens and Ones

Now let's expand the Roman numeral regular expression to cover the tens and ones place. This example shows the check for tens.

Example 7.7. Checking for Tens

```
>>> pattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)$'
>>> re.search(pattern, 'MCMXL')    ❶
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCML')     ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLX')    ❸
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXX')  ❹
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXX') ❺
>>>
```

❶ This matches the start of the string, then the first optional `M`, then `CM`, then `XL`, then the end of the string. Remember, the `(A|B|C)` syntax means "match exactly one of A, B, or C". You match `XL`, so you ignore the `XC` and `L?X?X?X?` choices, and then move on to the end of the string. `MCML` is the Roman numeral representation of `1940`.

❷ This matches the start of the string, then the first optional `M`, then `CM`, then `L?X?X?X?`. Of the `L?X?X?X?`, it matches the `L` and skips all three optional `X` characters. Then you move to the end of the string. `MCML` is the Roman numeral representation of `1950`.

❸ This matches the start of the string, then the first optional `M`, then `CM`, then the optional `L` and the first optional `X`, skips the second and third optional `X`, then the end of the string. `MCMLX` is the Roman numeral representation of `1960`.

❹ This matches the start of the string, then the first optional `M`, then `CM`, then the optional `L` and all three optional `X` characters, then the end of the string. `MCMLXXX` is the Roman numeral representation of `1980`.

❺ This matches the start of the string, then the first optional `M`, then `CM`, then the optional `L` and all three optional `X` characters, then *fails to match* the end of the string because there is still one more `X` unaccounted for. So the entire pattern fails to match, and returns `None`. `MCMLXXXX` is not a valid Roman numeral.

The expression for the ones place follows the same pattern. I'll spare you the details and show you the end result.

```
>>> pattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

So what does that look like using this alternate `{n,m}` syntax? This example shows the new syntax.

Example 7.8. Validating Roman Numerals with `{n,m}`

```
>>> pattern = '^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
>>> re.search(pattern, 'MDLV')             ❶
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMDCLXVI')         ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMMDCCCLXXXVIII') ❸
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'I')                ❹
<_sre.SRE_Match object at 0x008EEB48>
```

❶ This matches the start of the string, then one of a possible four `M` characters, then `D?C{0,3}`. Of that, it matches the optional `D` and zero of three possible `C` characters. Moving on, it matches `L?X{0,3}` by matching the optional `L` and zero of three possible `X` characters. Then it matches `V?I{0,3}` by matching the optional V and zero of three possible `I` characters, and finally the end of the string. `MDLV` is the Roman numeral representation of `1555`.

❷ This matches the start of the string, then two of a possible four `M` characters, then the `D?C{0,3}` with a `D` and one of three possible `C` characters; then `L?X{0,3}` with an `L` and one of three possible `X` characters; then `V?I{0,3}` with a `V` and one of three possible `I` characters; then the end of the string. `MMDCLXVI` is the

Roman numeral representation of `2666`.

❸ This matches the start of the string, then four out of four `M` characters, then `D?C{0,3}` with a `D` and three out of three `C` characters; then `L?X{0,3}` with an `L` and three out of three `X` characters; then `V?I{0,3}` with a `V` and three out of three `I` characters; then the end of the string. `MMMMDCCCLXXXVIII` is the Roman numeral representation of `3888`, and it's the longest Roman numeral you can write without extended syntax.

❹ Watch closely. (I feel like a magician. "Watch closely, kids, I'm going to pull a rabbit out of my hat.") This matches the start of the string, then zero out of four `M`, then matches `D?C{0,3}` by skipping the optional `D` and matching zero out of three `C`, then matches `L?X{0,3}` by skipping the optional `L` and matching zero out of three `X`, then matches `V?I{0,3}` by skipping the optional `V` and matching one out of three `I`. Then the end of the string. Whoa.

If you followed all that and understood it on the first try, you're doing better than I did. Now imagine trying to understand someone else's regular expressions, in the middle of a critical function of a large program. Or even imagine coming back to your own regular expressions a few months later. I've done it, and it's not a pretty sight.

In the next section you'll explore an alternate syntax that can help keep your expressions maintainable.

7.5. Verbose Regular Expressions

So far you've just been dealing with what I'll call "compact" regular expressions. As you've seen, they are difficult to read, and even if you figure out what one does, that's no guarantee that you'll be able to understand it six months later. What you really need is inline documentation.

Python allows you to do this with something called *verbose regular expressions*. A verbose regular expression is different from a compact regular expression in two ways:

- Whitespace is ignored. Spaces, tabs, and carriage returns are not matched as spaces, tabs, and carriage returns. They're not matched at all. (If you want to match a space in a verbose regular expression, you'll need to escape it by putting a backslash in front of it.)
- Comments are ignored. A comment in a verbose regular expression is just like a comment in Python code: it starts with a # character and goes until the end of the line. In this case it's a comment within a multi−line string instead of within your source code, but it works the same way.

This will be more clear with an example. Let's revisit the compact regular expression you've been working with, and make it a verbose regular expression. This example shows how.

Example 7.9. Regular Expressions with Inline Comments

```
>>> pattern = """
    ^                   # beginning of string
    M{0,4}              # thousands - 0 to 4 M's
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C's),
                        #            or 500-800 (D, followed by 0 to 3 C's)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
                        #        or 50-80 (L, followed by 0 to 3 X's)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
                        #        or 5-8 (V, followed by 0 to 3 I's)
    $                   # end of string
    """
>>> re.search(pattern, 'M', re.VERBOSE)                ❶
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXIX', re.VERBOSE)        ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMMDCCCLXXXVIII', re.VERBOSE) ❸
```

```
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'M')                    ④
```

❶ The most important thing to remember when using verbose regular expressions is that you need to pass an extra argument when working with them: `re.VERBOSE` is a constant defined in the `re` module that signals that the pattern should be treated as a verbose regular expression. As you can see, this pattern has quite a bit of whitespace (all of which is ignored), and several comments (all of which are ignored). Once you ignore the whitespace and the comments, this is exactly the same regular expression as you saw in the previous section, but it's a lot more readable.

❷ This matches the start of the string, then one of a possible four `M`, then `CM`, then `L` and three of a possible three `X`, then `IX`, then the end of the string.

❸ This matches the start of the string, then four of a possible four `M`, then `D` and three of a possible three `C`, then `L` and three of a possible three `X`, then `V` and three of a possible three `I`, then the end of the string.

❹ This does not match. Why? Because it doesn't have the `re.VERBOSE` flag, so the `re.search` function is treating the pattern as a compact regular expression, with significant whitespace and literal hash marks. Python can't auto−detect whether a regular expression is verbose or not. Python assumes every regular expression is compact unless you explicitly state that it is verbose.

7.6. Case study: Parsing Phone Numbers

So far you've concentrated on matching whole patterns. Either the pattern matches, or it doesn't. But regular expressions are much more powerful than that. When a regular expression *does* match, you can pick out specific pieces of it. You can find out what matched where.

This example came from another real−world problem I encountered, again from a previous day job. The problem: parsing an American phone number. The client wanted to be able to enter the number free−form (in a single field), but then wanted to store the area code, trunk, number, and optionally an extension separately in the company's database. I scoured the Web and found many examples of regular expressions that purported to do this, but none of them were permissive enough.

Here are the phone numbers I needed to be able to accept:

- `800−555−1212`
- `800 555 1212`
- `800.555.1212`
- `(800) 555−1212`
- `1−800−555−1212`
- `800−555−1212−1234`
- `800−555−1212x1234`
- `800−555−1212 ext. 1234`
- `work 1−(800) 555.1212 #1234`

Quite a variety! In each of these cases, I need to know that the area code was `800`, the trunk was `555`, and the rest of the phone number was `1212`. For those with an extension, I need to know that the extension was `1234`.

Let's work through developing a solution for phone number parsing. This example shows the first step.

Example 7.10. Finding Numbers

```
>>> phonePattern = re.compile(r'^(\d{3})−(\d{3})−(\d{4})$') ❶
>>> phonePattern.search('800−555−1212').groups()            ❷
('800', '555', '1212')
```

```
>>> phonePattern.search('800-555-1212-1234')                    ❸
>>>
```

❶ Always read regular expressions from left to right. This one matches the beginning of the string, and then `(\d{3})`. What's `\d{3}`? Well, the `{3}` means "match exactly three numeric digits"; it's a variation on the `{n,m} syntax` you saw earlier. `\d` means "any numeric digit" (`0` through `9`). Putting it in parentheses means "match exactly three numeric digits, *and then remember them as a group that I can ask for later*". Then match a literal hyphen. Then match another group of exactly three digits. Then another literal hyphen. Then another group of exactly four digits. Then match the end of the string.

❷ To get access to the groups that the regular expression parser remembered along the way, use the `groups()` method on the object that the `search` function returns. It will return a tuple of however many groups were defined in the regular expression. In this case, you defined three groups, one with three digits, one with three digits, and one with four digits.

❸ This regular expression is not the final answer, because it doesn't handle a phone number with an extension on the end. For that, you'll need to expand the regular expression.

Example 7.11. Finding the Extension

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$') ❶
>>> phonePattern.search('800-555-1212-1234').groups()             ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800 555 1212 1234')                      ❸
>>>
>>> phonePattern.search('800-555-1212')                           ❹
>>>
```

❶ This regular expression is almost identical to the previous one. Just as before, you match the beginning of the string, then a remembered group of three digits, then a hyphen, then a remembered group of three digits, then a hyphen, then a remembered group of four digits. What's new is that you then match another hyphen, and a remembered group of one or more digits, then the end of the string.

❷ The `groups()` method now returns a tuple of four elements, since the regular expression now defines four groups to remember.

❸ Unfortunately, this regular expression is not the final answer either, because it assumes that the different parts of the phone number are separated by hyphens. What if they're separated by spaces, or commas, or dots? You need a more general solution to match several different types of separators.

❹ Oops! Not only does this regular expression not do everything you want, it's actually a step backwards, because now you can't parse phone numbers *without* an extension. That's not what you wanted at all; if the extension is there, you want to know what it is, but if it's not there, you still want to know what the different parts of the main number are.

The next example shows the regular expression to handle separators between the different parts of the phone number.

Example 7.12. Handling Different Separators

```
>>> phonePattern = re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$') ❶
>>> phonePattern.search('800 555 1212 1234').groups()                   ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212-1234').groups()                   ❸
('800', '555', '1212', '1234')
>>> phonePattern.search('80055512121234')                               ❹
>>>
>>> phonePattern.search('800-555-1212')                                 ❺
>>>
```

1. Hang on to your hat. You're matching the beginning of the string, then a group of three digits, then `\D+`. What the heck is that? Well, `\D` matches any character *except* a numeric digit, and `+` means "1 or more". So `\D+` matches one or more characters that are not digits. This is what you're using instead of a literal hyphen, to try to match different separators.
2. Using `\D+` instead of `-` means you can now match phone numbers where the parts are separated by spaces instead of hyphens.
3. Of course, phone numbers separated by hyphens still work too.
4. Unfortunately, this is still not the final answer, because it assumes that there is a separator at all. What if the phone number is entered without any spaces or hyphens at all?
5. Oops! This still hasn't fixed the problem of requiring extensions. Now you have two problems, but you can solve both of them with the same technique.

The next example shows the regular expression for handling phone numbers *without* separators.

Example 7.13. Handling Numbers Without Separators

```
>>> phonePattern = re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ❶
>>> phonePattern.search('80055512121234').groups()                      ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800.555.1212 x1234').groups()                  ❸
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()                        ❹
('800', '555', '1212', '')
>>> phonePattern.search('(800)5551212 x1234')                           ❺
>>>
```

1. The only change you've made since that last step is changing all the `+` to `*`. Instead of `\D+` between the parts of the phone number, you now match on `\D*`. Remember that `+` means "1 or more"? Well, `*` means "zero or more". So now you should be able to parse phone numbers even when there is no separator character at all.
2. Lo and behold, it actually works. Why? You matched the beginning of the string, then a remembered group of three digits (`800`), then zero non-numeric characters, then a remembered group of three digits (`555`), then zero non-numeric characters, then a remembered group of four digits (`1212`), then zero non-numeric characters, then a remembered group of an arbitrary number of digits (`1234`), then the end of the string.
3. Other variations work now too: dots instead of hyphens, and both a space and an `x` before the extension.
4. Finally, you've solved the other long-standing problem: extensions are optional again. If no extension is found, the `groups()` method still returns a tuple of four elements, but the fourth element is just an empty string.
5. I hate to be the bearer of bad news, but you're not finished yet. What's the problem here? There's an extra character before the area code, but the regular expression assumes that the area code is the first thing at the beginning of the string. No problem, you can use the same technique of "zero or more non-numeric characters" to skip over the leading characters before the area code.

The next example shows how to handle leading characters in phone numbers.

Example 7.14. Handling Leading Characters

```
>>> phonePattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ❶
>>> phonePattern.search('(800)5551212 ext. 1234').groups()                 ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()                           ❸
('800', '555', '1212', '')
>>> phonePattern.search('work 1-(800) 555.1212 #1234')                     ❹
>>>
```

❶ This is the same as in the previous example, except now you're matching `\D*`, zero or more non−numeric characters, before the first remembered group (the area code). Notice that you're not remembering these non−numeric characters (they're not in parentheses). If you find them, you'll just skip over them and then start remembering the area code whenever you get to it.

❷ You can successfully parse the phone number, even with the leading left parenthesis before the area code. (The right parenthesis after the area code is already handled; it's treated as a non−numeric separator and matched by the `\D*` after the first remembered group.)

❸ Just a sanity check to make sure you haven't broken anything that used to work. Since the leading characters are entirely optional, this matches the beginning of the string, then zero non−numeric characters, then a remembered group of three digits (`800`), then one non−numeric character (the hyphen), then a remembered group of three digits (`555`), then one non−numeric character (the hyphen), then a remembered group of four digits (`1212`), then zero non−numeric characters, then a remembered group of zero digits, then the end of the string.

❹ This is where regular expressions make me want to gouge my eyes out with a blunt object. Why doesn't this phone number match? Because there's a `1` before the area code, but you assumed that all the leading characters before the area code were non−numeric characters (`\D*`). Aargh.

Let's back up for a second. So far the regular expressions have all matched from the beginning of the string. But now you see that there may be an indeterminate amount of stuff at the beginning of the string that you want to ignore. Rather than trying to match it all just so you can skip over it, let's take a different approach: don't explicitly match the beginning of the string at all. This approach is shown in the next example.

Example 7.15. Phone Number, Wherever I May Find Ye

```
>>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ❶
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()      ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')                               ❸
('800', '555', '1212', '')
>>> phonePattern.search('80055512121234')                             ❹
('800', '555', '1212', '1234')
```

❶ Note the lack of `^` in this regular expression. You are not matching the beginning of the string anymore. There's nothing that says you need to match the entire input with your regular expression. The regular expression engine will do the hard work of figuring out where the input string starts to match, and go from there.

❷ Now you can successfully parse a phone number that includes leading characters and a leading digit, plus any number of any kind of separators around each part of the phone number.

❸ Sanity check. this still works.

❹ That still works too.

See how quickly a regular expression can get out of control? Take a quick glance at any of the previous iterations. Can you tell the difference between one and the next?

While you still understand the final answer (and it is the final answer; if you've discovered a case it doesn't handle, I don't want to know about it), let's write it out as a verbose regular expression, before you forget why you made the choices you made.

Example 7.16. Parsing Phone Numbers (Final Version)

```
>>> phonePattern = re.compile(r'''
                # don't match beginning of string, number can start anywhere
    (\d{3})     # area code is 3 digits (e.g. '800')
```

```
    \D*          # optional separator is any number of non-digits
    (\d{3})      # trunk is 3 digits (e.g. '555')
    \D*          # optional separator
    (\d{4})      # rest of number is 4 digits (e.g. '1212')
    \D*          # optional separator
    (\d*)        # extension is optional and can be any number of digits
    $            # end of string
    ''', re.VERBOSE)
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()        ❶
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')                                ❷
('800', '555', '1212', '')
```

❶ Other than being spread out over multiple lines, this is exactly the same regular expression as the last step, so it's no surprise that it parses the same inputs.

❷ Final sanity check. Yes, this still works. You're done.

Further Reading on Regular Expressions

- Regular Expression HOWTO (http://py−howto.sourceforge.net/regex/regex.html) teaches about regular expressions and how to use them in Python.
- *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes the `re` module (http://www.python.org/doc/current/lib/module−re.html).

7.7. Summary

This is just the tiniest tip of the iceberg of what regular expressions can do. In other words, even though you're completely overwhelmed by them now, believe me, you ain't seen nothing yet.

You should now be familiar with the following techniques:

- `^` matches the beginning of a string.
- `$` matches the end of a string.
- `\b` matches a word boundary.
- `\d` matches any numeric digit.
- `\D` matches any non−numeric character.
- `x?` matches an optional `x` character (in other words, it matches an `x` zero or one times).
- `x*` matches `x` zero or more times.
- `x+` matches `x` one or more times.
- `x{n,m}` matches an `x` character at least `n` times, but not more than `m` times.
- `(a|b|c)` matches either `a` or `b` or `c`.
- `(x)` in general is a *remembered group*. You can get the value of what matched by using the `groups()` method of the object returned by `re.search`.

Regular expressions are extremely powerful, but they are not the correct solution for every problem. You should learn enough about them to know when they are appropriate, when they will solve your problems, and when they will cause more problems than they solve.

> Some people, when confronted with a problem, think "I know, I'll use regular expressions."
> Now they have two problems.
>
> −−Jamie Zawinski, in comp.emacs.xemacs (http://groups.google.com/groups?selm=33F0C496.370D7C45%40netscape.com)

Chapter 8. HTML Processing

8.1. Diving in

I often see questions on comp.lang.python (http://groups.google.com/groups?group=comp.lang.python) like "How can I list all the [headers|images|links] in my HTML document?" "How do I parse/translate/munge the text of my HTML document but leave the tags alone?" "How can I add/remove/quote attributes of all my HTML tags at once?" This chapter will answer all of these questions.

Here is a complete, working Python program in two parts. The first part, `BaseHTMLProcessor.py`, is a generic tool to help you process HTML files by walking through the tags and text blocks. The second part, `dialect.py`, is an example of how to use `BaseHTMLProcessor.py` to translate the text of an HTML document but leave the tags alone. Read the `doc string`s and comments to get an overview of what's going on. Most of it will seem like black magic, because it's not obvious how any of these class methods ever get called. Don't worry, all will be revealed in due time.

Example 8.1. `BaseHTMLProcessor.py`

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
from sgmllib import SGMLParser
import htmlentitydefs

class BaseHTMLProcessor(SGMLParser):
    def reset(self):
        # extend (called by SGMLParser.__init__)
        self.pieces = []
        SGMLParser.reset(self)

    def unknown_starttag(self, tag, attrs):
        # called for each start tag
        # attrs is a list of (attr, value) tuples
        # e.g. for <pre class="screen">, tag="pre", attrs=[("class", "screen")]
        # Ideally we would like to reconstruct original tag and attributes, but
        # we may end up quoting attribute values that weren't quoted in the source
        # document, or we may change the type of quotes around the attribute value
        # (single to double quotes).
        # Note that improperly embedded non−HTML code (like client−side Javascript)
        # may be parsed incorrectly by the ancestor, causing runtime script errors.
        # All non−HTML code must be enclosed in HTML comment tags (<!−− code −−>)
        # to ensure that it will pass through this parser unaltered (in handle_comment).
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())

    def unknown_endtag(self, tag):
        # called for each end tag, e.g. for </pre>, tag will be "pre"
        # Reconstruct the original end tag.
        self.pieces.append("</%(tag)s>" % locals())

    def handle_charref(self, ref):
        # called for each character reference, e.g. for " ", ref will be "160"
        # Reconstruct the original character reference.
        self.pieces.append("&#%(ref)s;" % locals())

    def handle_entityref(self, ref):
```

```
        # called for each entity reference, e.g. for "&copy;", ref will be "copy"
        # Reconstruct the original entity reference.
        self.pieces.append("&%(ref)s" % locals())
        # standard HTML entities are closed with a semicolon; other entities are not
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")

    def handle_data(self, text):
        # called for each block of plain text, i.e. outside of any tag and
        # not containing any character or entity references
        # Store the original text verbatim.
        self.pieces.append(text)

    def handle_comment(self, text):
        # called for each HTML comment, e.g. <!-- insert Javascript code here -->
        # Reconstruct the original comment.
        # It is especially important that the source document enclose client-side
        # code (like Javascript) within comments so it can pass through this
        # processor undisturbed; see comments in unknown_starttag for details.
        self.pieces.append("<!--%(text)s-->" % locals())

    def handle_pi(self, text):
        # called for each processing instruction, e.g. <?instruction>
        # Reconstruct original processing instruction.
        self.pieces.append("<?%(text)s>" % locals())

    def handle_decl(self, text):
        # called for the DOCTYPE, if present, e.g.
        # <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        #     "http://www.w3.org/TR/html4/loose.dtd">
        # Reconstruct original DOCTYPE
        self.pieces.append("<!%(text)s>" % locals())

    def output(self):
        """Return processed HTML as a single string"""
        return "".join(self.pieces)
```

Example 8.2. `dialect.py`

```
import re
from BaseHTMLProcessor import BaseHTMLProcessor

class Dialectizer(BaseHTMLProcessor):
    subs = ()

    def reset(self):
        # extend (called from __init__ in ancestor)
        # Reset all data attributes
        self.verbatim = 0
        BaseHTMLProcessor.reset(self)

    def start_pre(self, attrs):
        # called for every <pre> tag in HTML source
        # Increment verbatim mode count, then handle tag like normal
        self.verbatim += 1
        self.unknown_starttag("pre", attrs)

    def end_pre(self):
        # called for every </pre> tag in HTML source
        # Decrement verbatim mode count
        self.unknown_endtag("pre")
```

```
        self.verbatim -= 1

    def handle_data(self, text):
        # override
        # called for every block of text in HTML source
        # If in verbatim mode, save text unaltered;
        # otherwise process the text with a series of substitutions
        self.pieces.append(self.verbatim and text or self.process(text))

    def process(self, text):
        # called from handle_data
        # Process text block by performing series of regular expression
        # substitutions (actual substitions are defined in descendant)
        for fromPattern, toPattern in self.subs:
            text = re.sub(fromPattern, toPattern, text)
        return text

class ChefDialectizer(Dialectizer):
    """convert HTML to Swedish Chef-speak

    based on the classic chef.x, copyright (c) 1992, 1993 John Hagerman
    """
    subs = ((r'a([nu])', r'u\1'),
            (r'A([nu])', r'U\1'),
            (r'a\B', r'e'),
            (r'A\B', r'E'),
            (r'en\b', r'ee'),
            (r'\Bew', r'oo'),
            (r'\Be\b', r'e-a'),
            (r'\be', r'i'),
            (r'\bE', r'I'),
            (r'\Bf', r'ff'),
            (r'\Bir', r'ur'),
            (r'(\w*?)i(\w*?)$', r'\1ee\2'),
            (r'\bow', r'oo'),
            (r'\bo', r'oo'),
            (r'\bO', r'Oo'),
            (r'the', r'zee'),
            (r'The', r'Zee'),
            (r'th\b', r't'),
            (r'\Btion', r'shun'),
            (r'\Bu', r'oo'),
            (r'\BU', r'Oo'),
            (r'v', r'f'),
            (r'V', r'F'),
            (r'w', r'w'),
            (r'W', r'W'),
            (r'([a-z])[.]', r'\1.  Bork Bork Bork!'))

class FuddDialectizer(Dialectizer):
    """convert HTML to Elmer Fudd-speak"""
    subs = ((r'[rl]', r'w'),
            (r'qu', r'qw'),
            (r'th\b', r'f'),
            (r'th', r'd'),
            (r'n[.]', r'n, uh-hah-hah-hah.'))

class OldeDialectizer(Dialectizer):
    """convert HTML to mock Middle English"""
    subs = ((r'i([bcdfghjklmnpqrstvwxyz])e\b', r'y\1'),
            (r'i([bcdfghjklmnpqrstvwxyz])e', r'y\1\1e'),
            (r'ick\b', r'yk'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
```

```
            (r'e[ea]([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
            (r'([bcdfghjklmnpqrstvwxyz])y', r'\1ee'),
            (r'([bcdfghjklmnpqrstvwxyz])er', r'\1re'),
            (r'([aeiou])re\b', r'\1r'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'i\1e'),
            (r'tion\b', r'cioun'),
            (r'ion\b', r'ioun'),
            (r'aid', r'ayde'),
            (r'ai', r'ey'),
            (r'ay\b', r'y'),
            (r'ay', r'ey'),
            (r'ant', r'aunt'),
            (r'ea', r'ee'),
            (r'oa', r'oo'),
            (r'ue', r'e'),
            (r'oe', r'o'),
            (r'ou', r'ow'),
            (r'ow', r'ou'),
            (r'\bhe', r'hi'),
            (r've\b', r'veth'),
            (r'se\b', r'e'),
            (r"'s\b", r'es'),
            (r'ic\b', r'ick'),
            (r'ics\b', r'icc'),
            (r'ical\b', r'ick'),
            (r'tle\b', r'til'),
            (r'll\b', r'l'),
            (r'ould\b', r'olde'),
            (r'own\b', r'oune'),
            (r'un\b', r'onne'),
            (r'rry\b', r'rye'),
            (r'est\b', r'este'),
            (r'pt\b', r'pte'),
            (r'th\b', r'the'),
            (r'ch\b', r'che'),
            (r'ss\b', r'sse'),
            (r'([wybdp])\b', r'\1e'),
            (r'([rnt])\b', r'\1\1e'),
            (r'from', r'fro'),
            (r'when', r'whan'))

def translate(url, dialectName="chef"):
    """fetch URL and translate using dialect

    dialect in ("chef", "fudd", "olde")"""
    import urllib
    sock = urllib.urlopen(url)
    htmlSource = sock.read()
    sock.close()
    parserName = "%sDialectizer" % dialectName.capitalize()
    parserClass = globals()[parserName]
    parser = parserClass()
    parser.feed(htmlSource)
    parser.close()
    return parser.output()

def test(url):
    """test all dialects against URL"""
    for dialect in ("chef", "fudd", "olde"):
        outfile = "%s.html" % dialect
        fsock = open(outfile, "wb")
        fsock.write(translate(url, dialect))
        fsock.close()
```

```
        import webbrowser
        webbrowser.open_new(outfile)

if __name__ == "__main__":
    test("http://diveintopython.org/odbchelper_list.html")
```

Example 8.3. Output of `dialect.py`

Running this script will translate Section 3.2, Introducing Lists into mock Swedish Chef−speak (../native_data_types/chef.html) (from The Muppets), mock Elmer Fudd−speak (../native_data_types/fudd.html) (from Bugs Bunny cartoons), and mock Middle English (../native_data_types/olde.html) (loosely based on Chaucer's *The Canterbury Tales*). If you look at the HTML source of the output pages, you'll see that all the HTML tags and attributes are untouched, but the text between the tags has been "translated" into the mock language. If you look closer, you'll see that, in fact, only the titles and paragraphs were translated; the code listings and screen examples were left untouched.

```
<div class="abstract">
<p>Lists awe <span class="application">Pydon</span>'s wowkhowse datatype.
If youw onwy expewience wif wists is awways in
<span class="application">Visuaw Basic</span> ow (God fowbid) de datastowe
in <span class="application">Powewbuiwdew</span>, bwace youwsewf fow
<span class="application">Pydon</span> wists.</p>
</div>
```

8.2. Introducing `sgmllib.py`

HTML processing is broken into three steps: breaking down the HTML into its constituent pieces, fiddling with the pieces, and reconstructing the pieces into HTML again. The first step is done by `sgmllib.py`, a part of the standard Python library.

The key to understanding this chapter is to realize that HTML is not just text, it is structured text. The structure is derived from the more−or−less−hierarchical sequence of start tags and end tags. Usually you don't work with HTML this way; you work with it *textually* in a text editor, or *visually* in a web browser or web authoring tool. `sgmllib.py` presents HTML *structurally*.

`sgmllib.py` contains one important class: `SGMLParser`. `SGMLParser` parses HTML into useful pieces, like start tags and end tags. As soon as it succeeds in breaking down some data into a useful piece, it calls a method on itself based on what it found. In order to use the parser, you subclass the `SGMLParser` class and override these methods. This is what I meant when I said that it presents HTML *structurally*: the structure of the HTML determines the sequence of method calls and the arguments passed to each method.

`SGMLParser` parses HTML into 8 kinds of data, and calls a separate method for each of them:

Start tag
: An HTML tag that starts a block, like `<html>`, `<head>`, `<body>`, or `<pre>`, or a standalone tag like `<br>` or `<img>`. When it finds a start tag *`tagname`*, `SGMLParser` will look for a method called `start_`*`tagname`* or `do_`*`tagname`*. For instance, when it finds a `<pre>` tag, it will look for a `start_pre` or `do_pre` method. If found, `SGMLParser` calls this method with a list of the tag's attributes; otherwise, it calls `unknown_starttag` with the tag name and list of attributes.

End tag
: An HTML tag that ends a block, like `</html>`, `</head>`, `</body>`, or `</pre>`. When it finds an end tag, `SGMLParser` will look for a method called `end_`*`tagname`*. If found, `SGMLParser` calls this method, otherwise it calls `unknown_endtag` with the tag name.

Character reference
: An escaped character referenced by its decimal or hexadecimal equivalent, like ` `. When found, `SGMLParser` calls `handle_charref` with the text of the decimal or hexadecimal character equivalent.

Entity reference
: An HTML entity, like `©`. When found, `SGMLParser` calls `handle_entityref` with the name of the HTML entity.

Comment
: An HTML comment, enclosed in `<!-- ... -->`. When found, `SGMLParser` calls `handle_comment` with the body of the comment.

Processing instruction
: An HTML processing instruction, enclosed in `<? ... >`. When found, `SGMLParser` calls `handle_pi` with the body of the processing instruction.

Declaration
: An HTML declaration, such as a `DOCTYPE`, enclosed in `<! ... >`. When found, `SGMLParser` calls `handle_decl` with the body of the declaration.

Text data
: A block of text. Anything that doesn't fit into the other 7 categories. When found, `SGMLParser` calls `handle_data` with the text.

Python 2.0 had a bug where `SGMLParser` would not recognize declarations at all (`handle_decl` would never be called), which meant that `DOCTYPE`s were silently ignored. This is fixed in Python 2.1.

`sgmllib.py` comes with a test suite to illustrate this. You can run `sgmllib.py`, passing the name of an HTML file on the command line, and it will print out the tags and other elements as it parses them. It does this by subclassing the `SGMLParser` class and defining `unknown_starttag`, `unknown_endtag`, `handle_data` and other methods which simply print their arguments.

In the ActivePython IDE on Windows, you can specify command line arguments in the "Run script" dialog. Separate multiple arguments with spaces.

Example 8.4. Sample test of `sgmllib.py`

Here is a snippet from the table of contents of the HTML version of this book. Of course your paths may vary. (If you haven't downloaded the HTML version of the book, you can do so at http://diveintopython.org/.

```
c:\python23\lib> type "c:\downloads\diveintopython\html\toc\index.html"
 <!DOCTYPE html
  PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

      <title>Dive Into Python</title>
      <link rel="stylesheet" href="diveintopython.css" type="text/css">

... rest of file omitted for brevity ...
```

Running this through the test suite of `sgmllib.py` yields this output:

```
c:\python23\lib> python sgmllib.py "c:\downloads\diveintopython\html\toc\index.html"
data: '\n\n'
start tag: <html lang="en" >
data: '\n   '
```

```
start tag: <head>
data: '\n      '
start tag: <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" >
data: '\n   \n      '
start tag: <title>
data: 'Dive Into Python'
end tag: </title>
data: '\n      '
start tag: <link rel="stylesheet" href="diveintopython.css" type="text/css" >
data: '\n      '

... rest of output omitted for brevity ...
```

Here's the roadmap for the rest of the chapter:

- Subclass `SGMLParser` to create classes that extract interesting data out of HTML documents.
- Subclass `SGMLParser` to create `BaseHTMLProcessor`, which overrides all 8 handler methods and uses them to reconstruct the original HTML from the pieces.
- Subclass `BaseHTMLProcessor` to create `Dialectizer`, which adds some methods to process specific HTML tags specially, and overrides the `handle_data` method to provide a framework for processing the text blocks between the HTML tags.
- Subclass `Dialectizer` to create classes that define text processing rules used by `Dialectizer.handle_data`.
- Write a test suite that grabs a real web page from `http://diveintopython.org/` and processes it.

Along the way, you'll also learn about `locals`, `globals`, and dictionary−based string formatting.

8.3. Extracting data from HTML documents

To extract data from HTML documents, subclass the `SGMLParser` class and define methods for each tag or entity you want to capture.

The first step to extracting data from an HTML document is getting some HTML. If you have some HTML lying around on your hard drive, you can use file functions to read it, but the real fun begins when you get HTML from live web pages.

Example 8.5. Introducing `urllib`

```
>>> import urllib                                     ❶
>>> sock = urllib.urlopen("http://diveintopython.org/") ❷
>>> htmlSource = sock.read()                          ❸
>>> sock.close()                                      ❹
>>> print htmlSource                                  ❺
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"
      <meta http-equiv='Content-Type' content='text/html; charset=ISO-8859-1'>
   <title>Dive Into Python</title>
<link rel='stylesheet' href='diveintopython.css' type='text/css'>
<link rev='made' href='mailto:mark@diveintopython.org'>
<meta name='keywords' content='Python, Dive Into Python, tutorial, object-oriented, programming, docu
<meta name='description' content='a free Python tutorial for experienced programmers'>
</head>
<body bgcolor='white' text='black' link='#0000FF' vlink='#840084' alink='#0000FF'>
<table cellpadding='0' cellspacing='0' border='0' width='100%'>
<tr><td class='header' width='1%' valign='top'>diveintopython.org</td>
<td width='99%' align='right'><hr size='1' noshade></td></tr>
<tr><td class='tagline' colspan='2'>Python for experienced programmers</td></tr>
```

```
[...snip...]
```

1. The `urllib` module is part of the standard Python library. It contains functions for getting information about and actually retrieving data from Internet−based URLs (mainly web pages).
2. The simplest use of `urllib` is to retrieve the entire text of a web page using the `urlopen` function. Opening a URL is similar to opening a file. The return value of `urlopen` is a file−like object, which has some of the same methods as a file object.
3. The simplest thing to do with the file−like object returned by `urlopen` is `read`, which reads the entire HTML of the web page into a single string. The object also supports `readlines`, which reads the text line by line into a list.
4. When you're done with the object, make sure to `close` it, just like a normal file object.
5. You now have the complete HTML of the home page of `http://diveintopython.org/` in a string, and you're ready to parse it.

Example 8.6. Introducing `urllister.py`

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
from sgmllib import SGMLParser

class URLLister(SGMLParser):
    def reset(self):                              (1)
        SGMLParser.reset(self)
        self.urls = []

    def start_a(self, attrs):                     (2)
        href = [v for k, v in attrs if k=='href'] (3) (4)
        if href:
            self.urls.extend(href)
```

1. `reset` is called by the `__init__` method of `SGMLParser`, and it can also be called manually once an instance of the parser has been created. So if you need to do any initialization, do it in `reset`, not in `__init__`, so that it will be re−initialized properly when someone re−uses a parser instance.
2. `start_a` is called by `SGMLParser` whenever it finds an `<a>` tag. The tag may contain an `href` attribute, and/or other attributes, like `name` or `title`. The `attrs` parameter is a list of tuples, `[(attribute, value), (attribute, value), ...]`. Or it may be just an `<a>`, a valid (if useless) HTML tag, in which case `attrs` would be an empty list.
3. You can find out whether this `<a>` tag has an `href` attribute with a simple multi−variable list comprehension.
4. String comparisons like `k=='href'` are always case−sensitive, but that's safe in this case, because `SGMLParser` converts attribute names to lowercase while building `attrs`.

Example 8.7. Using `urllister.py`

```
>>> import urllib, urllister
>>> usock = urllib.urlopen("http://diveintopython.org/")
>>> parser = urllister.URLLister()
>>> parser.feed(usock.read())          (1)
>>> usock.close()                      (2)
>>> parser.close()                     (3)
>>> for url in parser.urls: print url  (4)
toc/index.html
#download
```

```
#languages
toc/index.html
appendix/history.html
download/diveintopython-html-5.0.zip
download/diveintopython-pdf-5.0.zip
download/diveintopython-word-5.0.zip
download/diveintopython-text-5.0.zip
download/diveintopython-html-flat-5.0.zip
download/diveintopython-xml-5.0.zip
download/diveintopython-common-5.0.zip

... rest of output omitted for brevity ...
```

1. Call the `feed` method, defined in `SGMLParser`, to get HTML into the parser.[1] It takes a string, which is what `usock.read()` returns.
2. Like files, you should `close` your URL objects as soon as you're done with them.
3. You should `close` your parser object, too, but for a different reason. You've read all the data and fed it to the parser, but the `feed` method isn't guaranteed to have actually processed all the HTML you give it; it may buffer it, waiting for more. Be sure to call `close` to flush the buffer and force everything to be fully parsed.
4. Once the parser is `close`d, the parsing is complete, and `parser.urls` contains a list of all the linked URLs in the HTML document. (Your output may look different, if the download links have been updated by the time you read this.)

8.4. Introducing `BaseHTMLProcessor.py`

`SGMLParser` doesn't produce anything by itself. It parses and parses and parses, and it calls a method for each interesting thing it finds, but the methods don't do anything. `SGMLParser` is an HTML *consumer*: it takes HTML and breaks it down into small, structured pieces. As you saw in the previous section, you can subclass `SGMLParser` to define classes that catch specific tags and produce useful things, like a list of all the links on a web page. Now you'll take this one step further by defining a class that catches everything `SGMLParser` throws at it and reconstructs the complete HTML document. In technical terms, this class will be an HTML *producer*.

`BaseHTMLProcessor` subclasses `SGMLParser` and provides all 8 essential handler methods: `unknown_starttag`, `unknown_endtag`, `handle_charref`, `handle_entityref`, `handle_comment`, `handle_pi`, `handle_decl`, and `handle_data`.

Example 8.8. Introducing `BaseHTMLProcessor`

```
class BaseHTMLProcessor(SGMLParser):
    def reset(self):                       ❶
        self.pieces = []
        SGMLParser.reset(self)

    def unknown_starttag(self, tag, attrs): ❷
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())

    def unknown_endtag(self, tag):         ❸
        self.pieces.append("</%(tag)s>" % locals())

    def handle_charref(self, ref):         ❹
        self.pieces.append("&#%(ref)s;" % locals())

    def handle_entityref(self, ref):       ❺
        self.pieces.append("&%(ref)s" % locals())
```

```
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")

    def handle_data(self, text):           ❻
        self.pieces.append(text)

    def handle_comment(self, text):        ❼
        self.pieces.append("<!--%(text)s-->" % locals())

    def handle_pi(self, text):             ❽
        self.pieces.append("<?%(text)s>" % locals())

    def handle_decl(self, text):
        self.pieces.append("<!%(text)s>" % locals())
```

❶ `reset`, called by `SGMLParser.__init__`, initializes `self.pieces` as an empty list before calling the ancestor method. `self.pieces` is a data attribute which will hold the pieces of the HTML document you're constructing. Each handler method will reconstruct the HTML that `SGMLParser` parsed, and each method will append that string to `self.pieces`. Note that `self.pieces` is a list. You might be tempted to define it as a string and just keep appending each piece to it. That would work, but Python is much more efficient at dealing with lists.[2]

❷ Since `BaseHTMLProcessor` does not define any methods for specific tags (like the `start_a` method in `URLLister`), `SGMLParser` will call `unknown_starttag` for every start tag. This method takes the tag (`tag`) and the list of attribute name/value pairs (`attrs`), reconstructs the original HTML, and appends it to `self.pieces`. The string formatting here is a little strange; you'll untangle that (and also the odd−looking `locals` function) later in this chapter.

❸ Reconstructing end tags is much simpler; just take the tag name and wrap it in the `</...>` brackets.

❹ When `SGMLParser` finds a character reference, it calls `handle_charref` with the bare reference. If the HTML document contains the reference ` `, `ref` will be `160`. Reconstructing the original complete character reference just involves wrapping `ref` in `&#...;` characters.

❺ Entity references are similar to character references, but without the hash mark. Reconstructing the original entity reference requires wrapping `ref` in `&...;` characters. (Actually, as an erudite reader pointed out to me, it's slightly more complicated than this. Only certain standard HTML entites end in a semicolon; other similar−looking entities do not. Luckily for us, the set of standard HTML entities is defined in a dictionary in a Python module called `htmlentitydefs`. Hence the extra `if` statement.)

❻ Blocks of text are simply appended to `self.pieces` unaltered.

❼ HTML comments are wrapped in `<!--...-->` characters.

❽ Processing instructions are wrapped in `<?...>` characters.

The HTML specification requires that all non−HTML (like client−side JavaScript) must be enclosed in HTML comments, but not all web pages do this properly (and all modern web browsers are forgiving if they don't). `BaseHTMLProcessor` is not forgiving; if script is improperly embedded, it will be parsed as if it were HTML. For instance, if the script contains less−than and equals signs, `SGMLParser` may incorrectly think that it has found tags and attributes. `SGMLParser` always converts tags and attribute names to lowercase, which may break the script, and `BaseHTMLProcessor` always encloses attribute values in double quotes (even if the original HTML document used single quotes or no quotes), which will certainly break the script. Always protect your client−side script within HTML comments.

Example 8.9. `BaseHTMLProcessor` output

```
    def output(self):               ❶
        """Return processed HTML as a single string"""
        return "".join(self.pieces) ❷
```

1. This is the one method in `BaseHTMLProcessor` that is never called by the ancestor `SGMLParser`. Since the other handler methods store their reconstructed HTML in `self.pieces`, this function is needed to join all those pieces into one string. As noted before, Python is great at lists and mediocre at strings, so you only create the complete string when somebody explicitly asks for it.
2. If you prefer, you could use the `join` method of the `string` module instead: `string.join(self.pieces, "")`

Further reading

- W3C (http://www.w3.org/) discusses character and entity references (http://www.w3.org/TR/REC−html40/charset.html#entities).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) confirms your suspicions that the `htmlentitydefs` module (http://www.python.org/doc/current/lib/module−htmlentitydefs.html) is exactly what it sounds like.

8.5. `locals` and `globals`

Let's digress from HTML processing for a minute and talk about how Python handles variables. Python has two built−in functions, `locals` and `globals`, which provide dictionary−based access to local and global variables.

Remember `locals`? You first saw it here:

```
def unknown_starttag(self, tag, attrs):
    strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
    self.pieces.append("<%(tag)s%(strattrs)s>" % locals())
```

No, wait, you can't learn about `locals` yet. First, you need to learn about namespaces. This is dry stuff, but it's important, so pay attention.

Python uses what are called namespaces to keep track of variables. A namespace is just like a dictionary where the keys are names of variables and the dictionary values are the values of those variables. In fact, you can access a namespace as a Python dictionary, as you'll see in a minute.

At any particular point in a Python program, there are several namespaces available. Each function has its own namespace, called the local namespace, which keeps track of the function's variables, including function arguments and locally defined variables. Each module has its own namespace, called the global namespace, which keeps track of the module's variables, including functions, classes, any other imported modules, and module−level variables and constants. And there is the built−in namespace, accessible from any module, which holds built−in functions and exceptions.

When a line of code asks for the value of a variable `x`, Python will search for that variable in all the available namespaces, in order:

1. local namespace − specific to the current function or class method. If the function defines a local variable `x`, or has an argument `x`, Python will use this and stop searching.
2. global namespace − specific to the current module. If the module has defined a variable, function, or class called `x`, Python will use that and stop searching.
3. built−in namespace − global to all modules. As a last resort, Python will assume that `x` is the name of built−in function or variable.

If Python doesn't find `x` in any of these namespaces, it gives up and raises a `NameError` with the message `There is no variable named 'x'`, which you saw back in Example 3.18, Referencing an Unbound Variable , but

you didn't appreciate how much work Python was doing before giving you that error.

Python 2.2 introduced a subtle but important change that affects the namespace search order: nested scopes. In versions of Python prior to 2.2, when you reference a variable within a nested function or `lambda` function, Python will search for that variable in the current (nested or `lambda`) function's namespace, then in the module's namespace. Python 2.2 will search for the variable in the current (nested or `lambda`) function's namespace, *then in the parent function's namespace*, then in the module's namespace. Python 2.1 can work either way; by default, it works like Python 2.0, but you can add the following line of code at the top of your module to make your module work like Python 2.2:

```
from __future__ import nested_scopes
```

Are you confused yet? Don't despair! This is really cool, I promise. Like many things in Python, namespaces are *directly accessible at run−time*. How? Well, the local namespace is accessible via the built−in `locals` function, and the global (module level) namespace is accessible via the built−in `globals` function.

Example 8.10. Introducing `locals`

```
>>> def foo(arg): ❶
...     x = 1
...     print locals()
...
>>> foo(7)          ❷
{'arg': 7, 'x': 1}
>>> foo('bar')      ❸
{'arg': 'bar', 'x': 1}
```

❶ The function `foo` has two variables in its local namespace: `arg`, whose value is passed in to the function, and `x`, which is defined within the function.

❷ `locals` returns a dictionary of name/value pairs. The keys of this dictionary are the names of the variables as strings; the values of the dictionary are the actual values of the variables. So calling `foo` with `7` prints the dictionary containing the function's two local variables: `arg` (`7`) and `x` (`1`).

❸ Remember, Python has dynamic typing, so you could just as easily pass a string in for `arg`; the function (and the call to `locals`) would still work just as well. `locals` works with all variables of all datatypes.

What `locals` does for the local (function) namespace, `globals` does for the global (module) namespace. `globals` is more exciting, though, because a module's namespace is more exciting.[3] Not only does the module's namespace include module−level variables and constants, it includes all the functions and classes defined in the module. Plus, it includes anything that was imported into the module.

Remember the difference between `from` *`module`* `import` and `import` *`module`*? With `import` *`module`*, the module itself is imported, but it retains its own namespace, which is why you need to use the module name to access any of its functions or attributes: *`module.function`*. But with `from` *`module`* `import`, you're actually importing specific functions and attributes from another module into your own namespace, which is why you access them directly without referencing the original module they came from. With the `globals` function, you can actually see this happen.

Example 8.11. Introducing `globals`

Look at the following block of code at the bottom of `BaseHTMLProcessor.py`:

```
if __name__ == "__main__":
```

```
    for k, v in globals().items():             ❶
        print k, "=", v
```

❶ Just so you don't get intimidated, remember that you've seen all this before. The `globals` function returns a dictionary, and you're iterating through the dictionary using the `items` method and multi−variable assignment. The only thing new here is the `globals` function.

Now running the script from the command line gives this output (note that your output may be slightly different, depending on your platform and where you installed Python):

```
c:\docbook\dip\py> python BaseHTMLProcessor.py

SGMLParser = sgmllib.SGMLParser                ❶
htmlentitydefs = <module 'htmlentitydefs' from 'C:\Python23\lib\htmlentitydefs.py'> ❷
BaseHTMLProcessor = __main__.BaseHTMLProcessor ❸
__name__ = __main__                            ❹
... rest of output omitted for brevity...
```

❶ `SGMLParser` was imported from `sgmllib`, using `from module import`. That means that it was imported directly into the module's namespace, and here it is.

❷ Contrast this with `htmlentitydefs`, which was imported using `import`. That means that the `htmlentitydefs` module itself is in the namespace, but the `entitydefs` variable defined within `htmlentitydefs` is not.

❸ This module only defines one class, `BaseHTMLProcessor`, and here it is. Note that the value here is the class itself, not a specific instance of the class.

❹ Remember the `if __name__` trick? When running a module (as opposed to importing it from another module), the built−in `__name__` attribute is a special value, `__main__`. Since you ran this module as a script from the command line, `__name__` is `__main__`, which is why the little test code to print the `globals` got executed.

Using the `locals` and `globals` functions, you can get the value of arbitrary variables dynamically, providing the variable name as a string. This mirrors the functionality of the `getattr` function, which allows you to access arbitrary functions dynamically by providing the function name as a string.

There is one other important difference between the `locals` and `globals` functions, which you should learn now before it bites you. It will bite you anyway, but at least then you'll remember learning it.

Example 8.12. `locals` is read−only, `globals` is not

```
def foo(arg):
    x = 1
    print locals()      ❶
    locals()["x"] = 2   ❷
    print "x=",x        ❸

z = 7
print "z=",z
foo(3)
globals()["z"] = 8      ❹
print "z=",z            ❺
```

❶ Since `foo` is called with `3`, this will print `{'arg': 3, 'x': 1}`. This should not be a surprise.

❷ `locals` is a function that returns a dictionary, and here you are setting a value in that dictionary. You might think that this would change the value of the local variable `x` to `2`, but it doesn't. `locals` does not actually return the local namespace, it returns a copy. So changing it does nothing to the value of the

variables in the local namespace.

3. This prints `x= 1`, not `x= 2`.
4. After being burned by `locals`, you might think that this *wouldn't* change the value of `z`, but it does. Due to internal differences in how Python is implemented (which I'd rather not go into, since I don't fully understand them myself), `globals` returns the actual global namespace, not a copy: the exact opposite behavior of `locals`. So any changes to the dictionary returned by `globals` directly affect your global variables.
5. This prints `z= 8`, not `z= 7`.

8.6. Dictionary−based string formatting

Why did you learn about `locals` and `globals`? So you can learn about dictionary−based string formatting. As you recall, regular string formatting provides an easy way to insert values into strings. Values are listed in a tuple and inserted in order into the string in place of each formatting marker. While this is efficient, it is not always the easiest code to read, especially when multiple values are being inserted. You can't simply scan through the string in one pass and understand what the result will be; you're constantly switching between reading the string and reading the tuple of values.

There is an alternative form of string formatting that uses dictionaries instead of tuples of values.

Example 8.13. Introducing dictionary−based string formatting

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> "%(pwd)s" % params                                 (1)
'secret'
>>> "%(pwd)s is not a good password for %(uid)s" % params (2)
'secret is not a good password for sa'
>>> "%(database)s of mind, %(database)s of body" % params (3)
'master of mind, master of body'
```

1. Instead of a tuple of explicit values, this form of string formatting uses a dictionary, `params`. And instead of a simple `%s` marker in the string, the marker contains a name in parentheses. This name is used as a key in the `params` dictionary and subsitutes the corresponding value, `secret`, in place of the `%(pwd)s` marker.
2. Dictionary−based string formatting works with any number of named keys. Each key must exist in the given dictionary, or the formatting will fail with a `KeyError`.
3. You can even specify the same key twice; each occurrence will be replaced with the same value.

So why would you use dictionary−based string formatting? Well, it does seem like overkill to set up a dictionary of keys and values simply to do string formatting in the next line; it's really most useful when you happen to have a dictionary of meaningful keys and values already. Like `locals`.

Example 8.14. Dictionary−based string formatting in `BaseHTMLProcessor.py`

```
    def handle_comment(self, text):
        self.pieces.append("<!--%(text)s-->" % locals()) (1)
```

1. Using the built−in `locals` function is the most common use of dictionary−based string formatting. It means that you can use the names of local variables within your string (in this case, `text`, which was passed to the class method as an argument) and each named variable will be replaced by its value. If `text` is `'Begin page footer'`, the string formatting `"<!--%(text)s-->" % locals()` will resolve to the string `'<!--Begin page footer-->'`.

Example 8.15. More dictionary−based string formatting

```
    def unknown_starttag(self, tag, attrs):
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs]) ❶
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())                  ❷
```

❶ When this method is called, attrs is a list of key/value tuples, just like the items of a dictionary, which means you can use multi−variable assignment to iterate through it. This should be a familiar pattern by now, but there's a lot going on here, so let's break it down:

a. Suppose attrs is [('href', 'index.html'), ('title', 'Go to home page')].
b. In the first round of the list comprehension, key will get 'href', and value will get 'index.html'.
c. The string formatting ' %s="%s"' % (key, value) will resolve to ' href="index.html"'. This string becomes the first element of the list comprehension's return value.
d. In the second round, key will get 'title', and value will get 'Go to home page'.
e. The string formatting will resolve to ' title="Go to home page"'.
f. The list comprehension returns a list of these two resolved strings, and strattrs will join both elements of this list together to form ' href="index.html" title="Go to home page"'.

❷ Now, using dictionary−based string formatting, you insert the value of tag and strattrs into a string. So if tag is 'a', the final result would be '<a href="index.html" title="Go to home page">', and that is what gets appended to self.pieces.

Using dictionary−based string formatting with locals is a convenient way of making complex string formatting expressions more readable, but it comes with a price. There is a slight performance hit in making the call to locals, since locals builds a copy of the local namespace.

8.7. Quoting attribute values

A common question on comp.lang.python (http://groups.google.com/groups?group=comp.lang.python) is "I have a bunch of HTML documents with unquoted attribute values, and I want to properly quote them all. How can I do this?"[4] (This is generally precipitated by a project manager who has found the HTML−is−a−standard religion joining a large project and proclaiming that all pages must validate against an HTML validator. Unquoted attribute values are a common violation of the HTML standard.) Whatever the reason, unquoted attribute values are easy to fix by feeding HTML through BaseHTMLProcessor.

BaseHTMLProcessor consumes HTML (since it's descended from SGMLParser) and produces equivalent HTML, but the HTML output is not identical to the input. Tags and attribute names will end up in lowercase, even if they started in uppercase or mixed case, and attribute values will be enclosed in double quotes, even if they started in single quotes or with no quotes at all. It is this last side effect that you can take advantage of.

Example 8.16. Quoting attribute values

```
>>> htmlSource = """        ❶
...     <html>
...     <head>
...     <title>Test page</title>
...     </head>
...     <body>
...     <ul>
...     <li><a href=index.html>Home</a></li>
...     <li><a href=toc.html>Table of contents</a></li>
...     <li><a href=history.html>Revision history</a></li>
```

```
...     </body>
...     </html>
...     """
>>> from BaseHTMLProcessor import BaseHTMLProcessor
>>> parser = BaseHTMLProcessor()
>>> parser.feed(htmlSource) ❷
>>> print parser.output()   ❸
<html>
<head>
<title>Test page</title>
</head>
<body>
<ul>
<li><a href="index.html">Home</a></li>
<li><a href="toc.html">Table of contents</a></li>
<li><a href="history.html">Revision history</a></li>
</body>
</html>
```

❶ Note that the attribute values of the `href` attributes in the `<a>` tags are not properly quoted. (Also note that you're using triple quotes for something other than a `doc string`. And directly in the IDE, no less. They're very useful.)

❷ Feed the parser.

❸ Using the `output` function defined in `BaseHTMLProcessor`, you get the output as a single string, complete with quoted attribute values. While this may seem anti−climactic, think about how much has actually happened here: `SGMLParser` parsed the entire HTML document, breaking it down into tags, refs, data, and so forth; `BaseHTMLProcessor` used those elements to reconstruct pieces of HTML (which are still stored in `parser.pieces`, if you want to see them); finally, you called `parser.output`, which joined all the pieces of HTML into one string.

8.8. Introducing `dialect.py`

`Dialectizer` is a simple (and silly) descendant of `BaseHTMLProcessor`. It runs blocks of text through a series of substitutions, but it makes sure that anything within a `<pre>...</pre>` block passes through unaltered.

To handle the `<pre>` blocks, you define two methods in `Dialectizer`: `start_pre` and `end_pre`.

Example 8.17. Handling specific tags

```
    def start_pre(self, attrs):             ❶
        self.verbatim += 1                  ❷
        self.unknown_starttag("pre", attrs) ❸

    def end_pre(self):                      ❹
        self.unknown_endtag("pre")          ❺
        self.verbatim -= 1                  ❻
```

❶ `start_pre` is called every time `SGMLParser` finds a `<pre>` tag in the HTML source. (In a minute, you'll see exactly how this happens.) The method takes a single parameter, `attrs`, which contains the attributes of the tag (if any). `attrs` is a list of key/value tuples, just like `unknown_starttag` takes.

❷ In the `reset` method, you initialize a data attribute that serves as a counter for `<pre>` tags. Every time you hit a `<pre>` tag, you increment the counter; every time you hit a `</pre>` tag, you'll decrement the counter. (You could just use this as a flag and set it to `1` and reset it to `0`, but it's just as easy to do it this way, and this handles the odd (but possible) case of nested `<pre>` tags.) In a minute, you'll see how this counter is put to good use.

❸

That's it, that's the only special processing you do for `<pre>` tags. Now you pass the list of attributes along to `unknown_starttag` so it can do the default processing.

❹ `end_pre` is called every time `SGMLParser` finds a `</pre>` tag. Since end tags can not contain attributes, the method takes no parameters.

❺ First, you want to do the default processing, just like any other end tag.

❻ Second, you decrement your counter to signal that this `<pre>` block has been closed.

At this point, it's worth digging a little further into `SGMLParser`. I've claimed repeatedly (and you've taken it on faith so far) that `SGMLParser` looks for and calls specific methods for each tag, if they exist. For instance, you just saw the definition of `start_pre` and `end_pre` to handle `<pre>` and `</pre>`. But how does this happen? Well, it's not magic, it's just good Python coding.

Example 8.18. `SGMLParser`

```
    def finish_starttag(self, tag, attrs):               ❶
        try:
            method = getattr(self, 'start_' + tag)       ❷
        except AttributeError:                           ❸
            try:
                method = getattr(self, 'do_' + tag)      ❹
            except AttributeError:
                self.unknown_starttag(tag, attrs)        ❺
                return -1
            else:
                self.handle_starttag(tag, method, attrs) ❻
                return 0
        else:
            self.stack.append(tag)
            self.handle_starttag(tag, method, attrs)
            return 1                                     ❼

    def handle_starttag(self, tag, method, attrs):
        method(attrs)                                    ❽
```

❶ At this point, `SGMLParser` has already found a start tag and parsed the attribute list. The only thing left to do is figure out whether there is a specific handler method for this tag, or whether you should fall back on the default method (`unknown_starttag`).

❷ The "magic" of `SGMLParser` is nothing more than your old friend, `getattr`. What you may not have realized before is that `getattr` will find methods defined in descendants of an object as well as the object itself. Here the object is `self`, the current instance. So if `tag` is `'pre'`, this call to `getattr` will look for a `start_pre` method on the current instance, which is an instance of the `Dialectizer` class.

❸ `getattr` raises an `AttributeError` if the method it's looking for doesn't exist in the object (or any of its descendants), but that's okay, because you wrapped the call to `getattr` inside a `try...except` block and explicitly caught the `AttributeError`.

❹ Since you didn't find a `start_xxx` method, you'll also look for a `do_xxx` method before giving up. This alternate naming scheme is generally used for standalone tags, like `<br>`, which have no corresponding end tag. But you can use either naming scheme; as you can see, `SGMLParser` tries both for every tag. (You shouldn't define both a `start_xxx` and `do_xxx` handler method for the same tag, though; only the `start_xxx` method will get called.)

❺ Another `AttributeError`, which means that the call to `getattr` failed with `do_xxx`. Since you found neither a `start_xxx` nor a `do_xxx` method for this tag, you catch the

exception and fall back on the default method, `unknown_starttag`.

6. Remember, `try...except` blocks can have an `else` clause, which is called if no exception is raised during the `try...except` block. Logically, that means that you *did* find a `do_xxx` method for this tag, so you're going to call it.
7. By the way, don't worry about these different return values; in theory they mean something, but they're never actually used. Don't worry about the `self.stack.append(tag)` either; `SGMLParser` keeps track internally of whether your start tags are balanced by appropriate end tags, but it doesn't do anything with this information either. In theory, you could use this module to validate that your tags were fully balanced, but it's probably not worth it, and it's beyond the scope of this chapter. You have better things to worry about right now.
8. `start_xxx` and `do_xxx` methods are not called directly; the tag, method, and attributes are passed to this function, `handle_starttag`, so that descendants can override it and change the way *all* start tags are dispatched. You don't need that level of control, so you just let this method do its thing, which is to call the method (`start_xxx` or `do_xxx`) with the list of attributes. Remember, `method` is a function, returned from `getattr`, and functions are objects. (I know you're getting tired of hearing it, and I promise I'll stop saying it as soon as I run out of ways to use it to my advantage.) Here, the function object is passed into this dispatch method as an argument, and this method turns around and calls the function. At this point, you don't need to know what the function is, what it's named, or where it's defined; the only thing you need to know about the function is that it is called with one argument, `attrs`.

Now back to our regularly scheduled program: `Dialectizer`. When you left, you were in the process of defining specific handler methods for `<pre>` and `</pre>` tags. There's only one thing left to do, and that is to process text blocks with the pre−defined substitutions. For that, you need to override the `handle_data` method.

Example 8.19. Overriding the `handle_data` method

```
    def handle_data(self, text):                                         ❶
        self.pieces.append(self.verbatim and text or self.process(text)) ❷
```

1. `handle_data` is called with only one argument, the text to process.
2. In the ancestor `BaseHTMLProcessor`, the `handle_data` method simply appended the text to the output buffer, `self.pieces`. Here the logic is only slightly more complicated. If you're in the middle of a `<pre>...</pre>` block, `self.verbatim` will be some value greater than `0`, and you want to put the text in the output buffer unaltered. Otherwise, you will call a separate method to process the substitutions, then put the result of that into the output buffer. In Python, this is a one−liner, using the `and−or` trick.

You're close to completely understanding `Dialectizer`. The only missing link is the nature of the text substitutions themselves. If you know any Perl, you know that when complex text substitutions are required, the only real solution is regular expressions. The classes later in `dialect.py` define a series of regular expressions that operate on the text between the HTML tags. But you just had a whole chapter on regular expressions. You don't really want to slog through regular expressions again, do you? God knows I don't. I think you've learned enough for one chapter.

8.9. Putting it all together

It's time to put everything you've learned so far to good use. I hope you were paying attention.

Example 8.20. The `translate` function, part 1

```
def translate(url, dialectName="chef"): ❶
    import urllib                       ❷
```

```
    sock = urllib.urlopen(url)          ❸
    htmlSource = sock.read()
    sock.close()
```

❶ The `translate` function has an optional argument `dialectName`, which is a string that specifies the dialect you'll be using. You'll see how this is used in a minute.

❷ Hey, wait a minute, there's an `import` statement in this function! That's perfectly legal in Python. You're used to seeing `import` statements at the top of a program, which means that the imported module is available anywhere in the program. But you can also import modules within a function, which means that the imported module is only available within the function. If you have a module that is only ever used in one function, this is an easy way to make your code more modular. (When you find that your weekend hack has turned into an 800−line work of art and decide to split it up into a dozen reusable modules, you'll appreciate this.)

❸ Now you get the source of the given URL.

Example 8.21. The `translate` function, part 2: curiouser and curiouser

```
    parserName = "%sDialectizer" % dialectName.capitalize() ❶
    parserClass = globals()[parserName]                     ❷
    parser = parserClass()                                  ❸
```

❶ `capitalize` is a string method you haven't seen before; it simply capitalizes the first letter of a string and forces everything else to lowercase. Combined with some string formatting, you've taken the name of a dialect and transformed it into the name of the corresponding Dialectizer class. If `dialectName` is the string `'chef'`, `parserName` will be the string `'ChefDialectizer'`.

❷ You have the name of a class as a string (`parserName`), and you have the global namespace as a dictionary (`globals`()). Combined, you can get a reference to the class which the string names. (Remember, classes are objects, and they can be assigned to variables just like any other object.) If `parserName` is the string `'ChefDialectizer'`, `parserClass` will be the class `ChefDialectizer`.

❸ Finally, you have a class object (`parserClass`), and you want an instance of the class. Well, you already know how to do that: call the class like a function. The fact that the class is being stored in a local variable makes absolutely no difference; you just call the local variable like a function, and out pops an instance of the class. If `parserClass` is the class `ChefDialectizer`, `parser` will be an instance of the class `ChefDialectizer`.

Why bother? After all, there are only 3 `Dialectizer` classes; why not just use a `case` statement? (Well, there's no `case` statement in Python, but why not just use a series of `if` statements?) One reason: extensibility. The `translate` function has absolutely no idea how many Dialectizer classes you've defined. Imagine if you defined a new `FooDialectizer` tomorrow; `translate` would work by passing `'foo'` as the `dialectName`.

Even better, imagine putting `FooDialectizer` in a separate module, and importing it with `from` *`module`* `import`. You've already seen that this includes it in `globals`(), so `translate` would still work without modification, even though `FooDialectizer` was in a separate file.

Now imagine that the name of the dialect is coming from somewhere outside the program, maybe from a database or from a user−inputted value on a form. You can use any number of server−side Python scripting architectures to dynamically generate web pages; this function could take a URL and a dialect name (both strings) in the query string of a web page request, and output the "translated" web page.

Finally, imagine a `Dialectizer` framework with a plug−in architecture. You could put each `Dialectizer` class in a separate file, leaving only the `translate` function in `dialect.py`. Assuming a consistent naming scheme, the `translate` function could dynamic import the appropiate class from the appropriate file, given nothing but the dialect name. (You haven't seen dynamic importing yet, but I promise to cover it in a later chapter.) To add a new

dialect, you would simply add an appropriately−named file in the plug−ins directory (like `foodialect.py` which contains the `FooDialectizer` class). Calling the `translate` function with the dialect name `'foo'` would find the module `foodialect.py`, import the class `FooDialectizer`, and away you go.

Example 8.22. The `translate` function, part 3

```
    parser.feed(htmlSource) ❶
    parser.close()          ❷
    return parser.output()  ❸
```

❶ After all that imagining, this is going to seem pretty boring, but the `feed` function is what does the entire transformation. You had the entire HTML source in a single string, so you only had to call `feed` once. However, you can call `feed` as often as you want, and the parser will just keep parsing. So if you were worried about memory usage (or you knew you were going to be dealing with very large HTML pages), you could set this up in a loop, where you read a few bytes of HTML and fed it to the parser. The result would be the same.

❷ Because `feed` maintains an internal buffer, you should always call the parser's `close` method when you're done (even if you fed it all at once, like you did). Otherwise you may find that your output is missing the last few bytes.

❸ Remember, `output` is the function you defined on `BaseHTMLProcessor` that joins all the pieces of output you've buffered and returns them in a single string.

And just like that, you've "translated" a web page, given nothing but a URL and the name of a dialect.

Further reading

- You thought I was kidding about the server−side scripting idea. So did I, until I found this web−based dialectizer (http://rinkworks.com/dialect/). Unfortunately, source code does not appear to be available.

8.10. Summary

Python provides you with a powerful tool, `sgmllib.py`, to manipulate HTML by turning its structure into an object model. You can use this tool in many different ways.

- parsing the HTML looking for something specific
- aggregating the results, like the URL lister
- altering the structure along the way, like the attribute quoter
- transforming the HTML into something else by manipulating the text while leaving the tags alone, like the `Dialectizer`

Along with these examples, you should be comfortable doing all of the following things:

- Using `locals`() and `globals`() to access namespaces
- Formatting strings using dictionary−based substitutions

---

[1] The technical term for a parser like `SGMLParser` is a *consumer*: it consumes HTML and breaks it down. Presumably, the name `feed` was chosen to fit into the whole "consumer" motif. Personally, it makes me think of an exhibit in the zoo where there's just a dark cage with no trees or plants or evidence of life of any kind, but if you stand perfectly still and look really closely you can make out two beady eyes staring back at you from the far left corner, but you convince yourself that that's just your mind playing tricks on you, and the only way you can tell that the whole thing isn't just an empty cage is a small innocuous sign on the railing that reads, "Do not feed the parser." But maybe

that's just me. In any event, it's an interesting mental image.

[2] The reason Python is better at lists than strings is that lists are mutable but strings are immutable. This means that appending to a list just adds the element and updates the index. Since strings can not be changed after they are created, code like `s = s + newpiece` will create an entirely new string out of the concatenation of the original and the new piece, then throw away the original string. This involves a lot of expensive memory management, and the amount of effort involved increases as the string gets longer, so doing `s = s + newpiece` in a loop is deadly. In technical terms, appending `n` items to a list is $O(n)$, while appending `n` items to a string is $O(n^2)$.

[3] I don't get out much.

[4] All right, it's not that common a question. It's not up there with "What editor should I use to write Python code?" (answer: Emacs) or "Is Python better or worse than Perl?" (answer: "Perl is worse than Python because people wanted it worse." −Larry Wall, 10/14/1998) But questions about HTML processing pop up in one form or another about once a month, and among those questions, this is a popular one.

Chapter 9. XML Processing

9.1. Diving in

These next two chapters are about XML processing in Python. It would be helpful if you already knew what an XML document looks like, that it's made up of structured tags to form a hierarchy of elements, and so on. If this doesn't make sense to you, there are many XML tutorials (http://directory.google.com/Top/Computers/Data_Formats/Markup_Languages/XML/Resources/FAQs,_Help,_and_Tutorials/) that can explain the basics.

If you're not particularly interested in XML, you should still read these chapters, which cover important topics like Python packages, Unicode, command line arguments, and how to use `getattr` for method dispatching.

Being a philosophy major is not required, although if you have ever had the misfortune of being subjected to the writings of Immanuel Kant, you will appreciate the example program a lot more than if you majored in something useful, like computer science.

There are two basic ways to work with XML. One is called SAX ("Simple API for XML"), and it works by reading the XML a little bit at a time and calling a method for each element it finds. (If you read Chapter 8, *HTML Processing*, this should sound familiar, because that's how the `sgmllib` module works.) The other is called DOM ("Document Object Model"), and it works by reading in the entire XML document at once and creating an internal representation of it using native Python classes linked in a tree structure. Python has standard modules for both kinds of parsing, but this chapter will only deal with using the DOM.

The following is a complete Python program which generates pseudo−random output based on a context−free grammar defined in an XML format. Don't worry yet if you don't understand what that means; you'll examine both the program's input and its output in more depth throughout these next two chapters.

Example 9.1. `kgp.py`

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
"""Kant Generator for Python

Generates mock philosophy based on a context−free grammar

Usage: python kgp.py [options] [source]

Options:
  −g ..., −−grammar=...   use specified grammar file or URL
  −h, −−help              show this help
  −d                      show debugging information while parsing

Examples:
  kgp.py                  generates several paragraphs of Kantian philosophy
  kgp.py −g husserl.xml   generates several paragraphs of Husserl
  kpg.py "<xref id='paragraph'/>"  generates a paragraph of Kant
  kgp.py template.xml     reads from template.xml to decide what to generate
"""
from xml.dom import minidom
import random
import toolbox
import sys
```

```
import getopt

_debug = 0

class NoSourceError(Exception): pass

class KantGenerator:
    """generates mock philosophy based on a context-free grammar"""

    def __init__(self, grammar, source=None):
        self.loadGrammar(grammar)
        self.loadSource(source and source or self.getDefaultSource())
        self.refresh()

    def _load(self, source):
        """load XML input source, return parsed XML document

        - a URL of a remote XML file ("http://diveintopython.org/kant.xml")
        - a filename of a local XML file ("~/diveintopython/common/py/kant.xml")
        - standard input ("-")
        - the actual XML document, as a string
        """
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc

    def loadGrammar(self, grammar):
        """load context-free grammar"""
        self.grammar = self._load(grammar)
        self.refs = {}
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref

    def loadSource(self, source):
        """load source"""
        self.source = self._load(source)

    def getDefaultSource(self):
        """guess default source of the current grammar

        The default source will be one of the <ref>s that is not
        cross-referenced.  This sounds complicated but it's not.
        Example: The default source for kant.xml is
        "<xref id='section'/>", because 'section' is the one <ref>
        that is not <xref>'d anywhere in the grammar.
        In most grammars, the default source will produce the
        longest (and most interesting) output.
        """
        xrefs = {}
        for xref in self.grammar.getElementsByTagName("xref"):
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
        if not standaloneXrefs:
            raise NoSourceError, "can't guess source, and no source specified"
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)

    def reset(self):
        """reset parser"""
        self.pieces = []
        self.capitalizeNextWord = 0
```

```
    def refresh(self):
        """reset output buffer, re-parse entire source file, and return output

        Since parsing involves a good deal of randomness, this is an
        easy way to get new output without having to reload a grammar file
        each time.
        """
        self.reset()
        self.parse(self.source)
        return self.output()

    def output(self):
        """output generated text"""
        return "".join(self.pieces)

    def randomChildElement(self, node):
        """choose a random child element of a node

        This is a utility method used by do_xref and do_choice.
        """
        choices = [e for e in node.childNodes
                   if e.nodeType == e.ELEMENT_NODE]
        chosen = random.choice(choices)
        if _debug:
            sys.stderr.write('%s available choices: %s\n' % \
                (len(choices), [e.toxml() for e in choices]))
            sys.stderr.write('Chosen: %s\n' % chosen.toxml())
        return chosen

    def parse(self, node):
        """parse a single XML node

        A parsed XML document (from minidom.parse) is a tree of nodes
        of various types.  Each node is represented by an instance of the
        corresponding Python class (Element for a tag, Text for
        text data, Document for the top-level document).  The following
        statement constructs the name of a class method based on the type
        of node we're parsing ("parse_Element" for an Element node,
        "parse_Text" for a Text node, etc.) and then calls the method.
        """
        parseMethod = getattr(self, "parse_%s" % node.__class__.__name__)
        parseMethod(node)

    def parse_Document(self, node):
        """parse the document node

        The document node by itself isn't interesting (to us), but
        its only child, node.documentElement, is: it's the root node
        of the grammar.
        """
        self.parse(node.documentElement)

    def parse_Text(self, node):
        """parse a text node

        The text of a text node is usually added to the output buffer
        verbatim.  The one exception is that <p class='sentence'> sets
        a flag to capitalize the first letter of the next word.  If
        that flag is set, we capitalize the text and reset the flag.
        """
        text = node.data
        if self.capitalizeNextWord:
            self.pieces.append(text[0].upper())
```

```
            self.pieces.append(text[1:])
            self.capitalizeNextWord = 0
        else:
            self.pieces.append(text)

    def parse_Element(self, node):
        """parse an element

        An XML element corresponds to an actual tag in the source:
        <xref id='...'>, <p chance='...'>, <choice>, etc.
        Each element type is handled in its own method.  Like we did in
        parse(), we construct a method name based on the name of the
        element ("do_xref" for an <xref> tag, etc.) and
        call the method.
        """
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)

    def parse_Comment(self, node):
        """parse a comment

        The grammar can contain XML comments, but we ignore them
        """
        pass

    def do_xref(self, node):
        """handle <xref id='...'> tag

        An <xref id='...'> tag is a cross-reference to a <ref id='...'>
        tag.  <xref id='sentence'/> evaluates to a randomly chosen child of
        <ref id='sentence'>.
        """
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))

    def do_p(self, node):
        """handle <p> tag

        The <p> tag is the core of the grammar.  It can contain almost
        anything: freeform text, <choice> tags, <xref> tags, even other
        <p> tags.  If a "class='sentence'" attribute is found, a flag
        is set and the next word will be capitalized.  If a "chance='X'"
        attribute is found, there is an X% chance that the tag will be
        evaluated (and therefore a (100-X)% chance that it will be
        completely ignored)
        """
        keys = node.attributes.keys()
        if "class" in keys:
            if node.attributes["class"].value == "sentence":
                self.capitalizeNextWord = 1
        if "chance" in keys:
            chance = int(node.attributes["chance"].value)
            doit = (chance > random.randrange(100))
        else:
            doit = 1
        if doit:
            for child in node.childNodes: self.parse(child)

    def do_choice(self, node):
        """handle <choice> tag

        A <choice> tag contains one or more <p> tags.  One <p> tag
        is chosen at random and evaluated; the rest are ignored.
```

```
        """
        self.parse(self.randomChildElement(node))

def usage():
    print __doc__

def main(argv):
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage()
            sys.exit()
        elif opt == '-d':
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"):
            grammar = arg

    source = "".join(args)

    k = KantGenerator(grammar, source)
    print k.output()

if __name__ == "__main__":
    main(sys.argv[1:])
```

Example 9.2. `toolbox.py`

```
"""Miscellaneous utility functions"""

def openAnything(source):
    """URI, filename, or string --> stream

    This function lets you define parsers that take any input source
    (URL, pathname to local or network file, or actual data as a string)
    and deal with it in a uniform manner.  Returned object is guaranteed
    to have all the basic stdio read methods (read, readline, readlines).
    Just .close() the object when you're done with it.

    Examples:
    >>> from xml.dom import minidom
    >>> sock = openAnything("http://localhost/kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("c:\\inetpub\\wwwroot\\kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("<ref id='conjunction'><text>and</text><text>or</text></ref>")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    """
    if hasattr(source, "read"):
        return source

    if source == '-':
        import sys
```

```
        return sys.stdin

    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
        return urllib.urlopen(source)
    except (IOError, OSError):
        pass

    # try to open with native open function (if source is pathname)
    try:
        return open(source)
    except (IOError, OSError):
        pass

    # treat source as string
    import StringIO
    return StringIO.StringIO(str(source))
```

Run the program `kgp.py` by itself, and it will parse the default XML−based grammar, in `kant.xml`, and print several paragraphs worth of philosophy in the style of Immanuel Kant.

Example 9.3. Sample output of `kgp.py`

```
[you@localhost kgp]$ python kgp.py
     As is shown in the writings of Hume, our a priori concepts, in
reference to ends, abstract from all content of knowledge; in the study
of space, the discipline of human reason, in accordance with the
principles of philosophy, is the clue to the discovery of the
Transcendental Deduction.  The transcendental aesthetic, in all
theoretical sciences, occupies part of the sphere of human reason
concerning the existence of our ideas in general; still, the
never−ending regress in the series of empirical conditions constitutes
the whole content for the transcendental unity of apperception.  What
we have alone been able to show is that, even as this relates to the
architectonic of human reason, the Ideal may not contradict itself, but
it is still possible that it may be in contradictions with the
employment of the pure employment of our hypothetical judgements, but
natural causes (and I assert that this is the case) prove the validity
of the discipline of pure reason.  As we have already seen, time (and
it is obvious that this is true) proves the validity of time, and the
architectonic of human reason, in the full sense of these terms,
abstracts from all content of knowledge.  I assert, in the case of the
discipline of practical reason, that the Antinomies are just as
necessary as natural causes, since knowledge of the phenomena is a
posteriori.
    The discipline of human reason, as I have elsewhere shown, is by
its very nature contradictory, but our ideas exclude the possibility of
the Antinomies.  We can deduce that, on the contrary, the pure
employment of philosophy, on the contrary, is by its very nature
contradictory, but our sense perceptions are a representation of, in
the case of space, metaphysics.  The thing in itself is a
representation of philosophy.  Applied logic is the clue to the
discovery of natural causes.  However, what we have alone been able to
show is that our ideas, in other words, should only be used as a canon
for the Ideal, because of our necessary ignorance of the conditions.

[...snip...]
```

This is, of course, complete gibberish. Well, not complete gibberish. It is syntactically and grammatically correct (although very verbose −− Kant wasn't what you would call a get−to−the−point kind of guy). Some of it may actually be true (or at least the sort of thing that Kant would have agreed with), some of it is blatantly false, and most of it is simply incoherent. But all of it is in the style of Immanuel Kant.

Let me repeat that this is much, much funnier if you are now or have ever been a philosophy major.

The interesting thing about this program is that there is nothing Kant−specific about it. All the content in the previous example was derived from the grammar file, `kant.xml`. If you tell the program to use a different grammar file (which you can specify on the command line), the output will be completely different.

Example 9.4. Simpler output from `kgp.py`

```
[you@localhost kgp]$ python kgp.py −g binary.xml
00101001
[you@localhost kgp]$ python kgp.py −g binary.xml
10110100
```

You will take a closer look at the structure of the grammar file later in this chapter. For now, all you need to know is that the grammar file defines the structure of the output, and the `kgp.py` program reads through the grammar and makes random decisions about which words to plug in where.

9.2. Packages

Actually parsing an XML document is very simple: one line of code. However, before you get to that line of code, you need to take a short detour to talk about packages.

Example 9.5. Loading an XML document (a sneak peek)

```
>>> from xml.dom import minidom ❶
>>> xmldoc = minidom.parse('~/diveintopython/common/py/kgp/binary.xml')
```

❶ This is a syntax you haven't seen before. It looks almost like the `from` *`module`* `import` you know and love, but the `"."` gives it away as something above and beyond a simple import. In fact, `xml` is what is known as a package, `dom` is a nested package within `xml`, and `minidom` is a module within `xml.dom`.

That sounds complicated, but it's really not. Looking at the actual implementation may help. Packages are little more than directories of modules; nested packages are subdirectories. The modules within a package (or a nested package) are still just `.py` files, like always, except that they're in a subdirectory instead of the main `lib/` directory of your Python installation.

Example 9.6. File layout of a package

```
Python21/           root Python installation (home of the executable)
|
+−−lib/             library directory (home of the standard library modules)
   |
   +−− xml/         xml package (really just a directory with other stuff in it)
       |
       +−−sax/      xml.sax package (again, just a directory)
       |
       +−−dom/      xml.dom package (contains minidom.py)
       |
```

```
        +--parsers/  xml.parsers package (used internally)
```

So when you say `from xml.dom import minidom`, Python figures out that that means "look in the `xml` directory for a `dom` directory, and look in *that* for the `minidom` module, and import it as `minidom`". But Python is even smarter than that; not only can you import entire modules contained within a package, you can selectively import specific classes or functions from a module contained within a package. You can also import the package itself as a module. The syntax is all the same; Python figures out what you mean based on the file layout of the package, and automatically does the right thing.

Example 9.7. Packages are modules, too

```
>>> from xml.dom import minidom         ❶
>>> minidom
<module 'xml.dom.minidom' from 'C:\Python21\lib\xml\dom\minidom.pyc'>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml.dom.minidom import Element ❷
>>> Element
<class xml.dom.minidom.Element at 01095744>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml import dom                 ❸
>>> dom
<module 'xml.dom' from 'C:\Python21\lib\xml\dom\__init__.pyc'>
>>> import xml                          ❹
>>> xml
<module 'xml' from 'C:\Python21\lib\xml\__init__.pyc'>
```

❶ Here you're importing a module (`minidom`) from a nested package (`xml.dom`). The result is that `minidom` is imported into your namespace, and in order to reference classes within the `minidom` module (like `Element`), you need to preface them with the module name.

❷ Here you are importing a class (`Element`) from a module (`minidom`) from a nested package (`xml.dom`). The result is that `Element` is imported directly into your namespace. Note that this does not interfere with the previous import; the `Element` class can now be referenced in two ways (but it's all still the same class).

❸ Here you are importing the `dom` package (a nested package of `xml`) as a module in and of itself. Any level of a package can be treated as a module, as you'll see in a moment. It can even have its own attributes and methods, just the modules you've seen before.

❹ Here you are importing the root level `xml` package as a module.

So how can a package (which is just a directory on disk) be imported and treated as a module (which is always a file on disk)? The answer is the magical `__init__.py` file. You see, packages are not simply directories; they are directories with a specific file, `__init__.py`, inside. This file defines the attributes and methods of the package. For instance, `xml.dom` contains a `Node` class, which is defined in `xml/dom/__init__.py`. When you import a package as a module (like `dom` from `xml`), you're really importing its `__init__.py` file.

A package is a directory with the special `__init__.py` file in it. The `__init__.py` file defines the attributes and methods of the package. It doesn't need to define anything; it can just be an empty file, but it has to exist. But if `__init__.py` doesn't exist, the directory is just a directory, not a package, and it can't be imported or contain modules or nested packages.

So why bother with packages? Well, they provide a way to logically group related modules. Instead of having an `xml` package with `sax` and `dom` packages inside, the authors could have chosen to put all the `sax` functionality in `xmlsax.py` and all the `dom` functionality in `xmldom.py`, or even put all of it in a single module. But that would

have been unwieldy (as of this writing, the XML package has over 3000 lines of code) and difficult to manage (separate source files mean multiple people can work on different areas simultaneously).

If you ever find yourself writing a large subsystem in Python (or, more likely, when you realize that your small subsystem has grown into a large one), invest some time designing a good package architecture. It's one of the many things Python is good at, so take advantage of it.

9.3. Parsing XML

As I was saying, actually parsing an XML document is very simple: one line of code. Where you go from there is up to you.

Example 9.8. Loading an XML document (for real this time)

```
>>> from xml.dom import minidom                                         ❶
>>> xmldoc = minidom.parse('~/diveintopython/common/py/kgp/binary.xml') ❷
>>> xmldoc                                                              ❸
<xml.dom.minidom.Document instance at 010BE87C>
>>> print xmldoc.toxml()                                                ❹
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

❶ As you saw in the previous section, this imports the `minidom` module from the `xml.dom` package.

❷ Here is the one line of code that does all the work: `minidom.parse` takes one argument and returns a parsed representation of the XML document. The argument can be many things; in this case, it's simply a filename of an XML document on my local disk. (To follow along, you'll need to change the path to point to your downloaded examples directory.) But you can also pass a file object, or even a file−like object. You'll take advantage of this flexibility later in this chapter.

❸ The object returned from `minidom.parse` is a `Document` object, a descendant of the `Node` class. This `Document` object is the root level of a complex tree−like structure of interlocking Python objects that completely represent the XML document you passed to `minidom.parse`.

❹ `toxml` is a method of the `Node` class (and is therefore available on the `Document` object you got from `minidom.parse`). `toxml` prints out the XML that this `Node` represents. For the `Document` node, this prints out the entire XML document.

Now that you have an XML document in memory, you can start traversing through it.

Example 9.9. Getting child nodes

```
>>> xmldoc.childNodes    ❶
[<DOM Element: grammar at 17538908>]
>>> xmldoc.childNodes[0] ❷
<DOM Element: grammar at 17538908>
>>> xmldoc.firstChild    ❸
<DOM Element: grammar at 17538908>
```

❶ Every `Node` has a `childNodes` attribute, which is a list of the `Node` objects. A `Document` always has only one child node, the root element of the XML document (in this case, the `grammar` element).

❷ To get the first (and in this case, the only) child node, just use regular list syntax. Remember, there is nothing special going on here; this is just a regular Python list of regular Python objects.

❸ Since getting the first child node of a node is a useful and common activity, the `Node` class has a `firstChild` attribute, which is synonymous with `childNodes[0]`. (There is also a `lastChild` attribute, which is synonymous with `childNodes[-1]`.)

Example 9.10. `toxml` works on any node

```
>>> grammarNode = xmldoc.firstChild
>>> print grammarNode.toxml() ❶
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

❶ Since the `toxml` method is defined in the `Node` class, it is available on any XML node, not just the `Document` element.

Example 9.11. Child nodes can be text

```
>>> grammarNode.childNodes                    ❶
[<DOM Text node "\n">, <DOM Element: ref at 17533332>, \
<DOM Text node "\n">, <DOM Element: ref at 17549660>, <DOM Text node "\n">]
>>> print grammarNode.firstChild.toxml()      ❷



>>> print grammarNode.childNodes[1].toxml() ❸
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print grammarNode.childNodes[3].toxml() ❹
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
>>> print grammarNode.lastChild.toxml()       ❺


```

❶ Looking at the XML in binary.xml, you might think that the grammar has only two child nodes, the two ref elements. But you're missing something: the carriage returns! After the '<grammar>' and before the first '<ref>' is a carriage return, and this text counts as a child node of the grammar element. Similarly, there is a carriage return after each '</ref>'; these also count as child nodes. So grammar.childNodes is actually a list of 5 objects: 3 Text objects and 2 Element objects.

❷ The first child is a Text object representing the carriage return after the '<grammar>' tag and before the first '<ref>' tag.

❸ The second child is an `Element` object representing the first `ref` element.

❹ The fourth child is an `Element` object representing the second `ref` element.

❺ The last child is a `Text` object representing the carriage return after the `'</ref>'` end tag and before the `'</grammar>'` end tag.

Example 9.12. Drilling down all the way to text

```
>>> grammarNode
<DOM Element: grammar at 19167148>
>>> refNode = grammarNode.childNodes[1] ❶
>>> refNode
<DOM Element: ref at 17987740>
>>> refNode.childNodes                  ❷
[<DOM Text node "\n">, <DOM Text node "  ">, <DOM Element: p at 19315844>, \
<DOM Text node "\n">, <DOM Text node "  ">, \
<DOM Element: p at 19462036>, <DOM Text node "\n">]
>>> pNode = refNode.childNodes[2]
>>> pNode
<DOM Element: p at 19315844>
>>> print pNode.toxml()                 ❸
<p>0</p>
>>> pNode.firstChild                    ❹
<DOM Text node "0">
>>> pNode.firstChild.data               ❺
u'0'
```

❶ As you saw in the previous example, the first `ref` element is `grammarNode.childNodes[1]`, since childNodes[0] is a `Text` node for the carriage return.

❷ The `ref` element has its own set of child nodes, one for the carriage return, a separate one for the spaces, one for the `p` element, and so forth.

❸ You can even use the `toxml` method here, deeply nested within the document.

❹ The `p` element has only one child node (you can't tell that from this example, but look at `pNode.childNodes` if you don't believe me), and it is a `Text` node for the single character `'0'`.

❺ The `.data` attribute of a `Text` node gives you the actual string that the text node represents. But what is that `'u'` in front of the string? The answer to that deserves its own section.

9.4. Unicode

Unicode is a system to represent characters from all the world's different languages. When Python parses an XML document, all data is stored in memory as unicode.

You'll get to all that in a minute, but first, some background.

Historical note. Before unicode, there were separate character encoding systems for each language, each using the same numbers (0−255) to represent that language's characters. Some languages (like Russian) have multiple conflicting standards about how to represent the same characters; other languages (like Japanese) have so many characters that they require multiple−byte character sets. Exchanging documents between systems was difficult because there was no way for a computer to tell for certain which character encoding scheme the document author had used; the computer only saw numbers, and the numbers could mean different things. Then think about trying to store these documents in the same place (like in the same database table); you would need to store the character encoding alongside each piece of text, and make sure to pass it around whenever you passed the text around. Then think about

multilingual documents, with characters from multiple languages in the same document. (They typically used escape codes to switch modes; poof, you're in Russian koi8−r mode, so character 241 means this; poof, now you're in Mac Greek mode, so character 241 means something else. And so on.) These are the problems which unicode was designed to solve.

To solve these problems, unicode represents each character as a 2−byte number, from 0 to 65535.[5] Each 2−byte number represents a unique character used in at least one of the world's languages. (Characters that are used in multiple languages have the same numeric code.) There is exactly 1 number per character, and exactly 1 character per number. Unicode data is never ambiguous.

Of course, there is still the matter of all these legacy encoding systems. 7−bit ASCII, for instance, which stores English characters as numbers ranging from 0 to 127. (65 is capital "A", 97 is lowercase "a", and so forth.) English has a very simple alphabet, so it can be completely expressed in 7−bit ASCII. Western European languages like French, Spanish, and German all use an encoding system called ISO−8859−1 (also called "latin−1"), which uses the 7−bit ASCII characters for the numbers 0 through 127, but then extends into the 128−255 range for characters like n−with−a−tilde−over−it (241), and u−with−two−dots−over−it (252). And unicode uses the same characters as 7−bit ASCII for 0 through 127, and the same characters as ISO−8859−1 for 128 through 255, and then extends from there into characters for other languages with the remaining numbers, 256 through 65535.

When dealing with unicode data, you may at some point need to convert the data back into one of these other legacy encoding systems. For instance, to integrate with some other computer system which expects its data in a specific 1−byte encoding scheme, or to print it to a non−unicode−aware terminal or printer. Or to store it in an XML document which explicitly specifies the encoding scheme.

And on that note, let's get back to Python.

Python has had unicode support throughout the language since version 2.0. The XML package uses unicode to store all parsed XML data, but you can use unicode anywhere.

Example 9.13. Introducing unicode

```
>>> s = u'Dive in'            ❶
>>> s
u'Dive in'
>>> print s                   ❷
Dive in
```

❶ To create a unicode string instead of a regular ASCII string, add the letter "u" before the string. Note that this particular string doesn't have any non−ASCII characters. That's fine; unicode is a superset of ASCII (a very large superset at that), so any regular ASCII string can also be stored as unicode.

❷ When printing a string, Python will attempt to convert it to your default encoding, which is usually ASCII. (More on this in a minute.) Since this unicode string is made up of characters that are also ASCII characters, printing it has the same result as printing a normal ASCII string; the conversion is seamless, and if you didn't know that s was a unicode string, you'd never notice the difference.

Example 9.14. Storing non−ASCII characters

```
>>> s = u'La Pe\xf1a'         ❶
>>> print s                   ❷
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> print s.encode('latin-1') ❸
```

```
La Peña
```

1. The real advantage of unicode, of course, is its ability to store non−ASCII characters, like the Spanish "ñ" (n with a tilde over it). The unicode character code for the tilde−n is `0xf1` in hexadecimal (241 in decimal), which you can type like this: `\xf1`.
2. Remember I said that the `print` function attempts to convert a unicode string to ASCII so it can print it? Well, that's not going to work here, because your unicode string contains non−ASCII characters, so Python raises a `UnicodeError` error.
3. Here's where the conversion−from−unicode−to−other−encoding−schemes comes in. `s` is a unicode string, but `print` can only print a regular string. To solve this problem, you call the `encode` method, available on every unicode string, to convert the unicode string to a regular string in the given encoding scheme, which you pass as a parameter. In this case, you're using `latin−1` (also known as `iso−8859−1`), which includes the tilde−n (whereas the default ASCII encoding scheme did not, since it only includes characters numbered 0 through 127).

Remember I said Python usually converted unicode to ASCII whenever it needed to make a regular string out of a unicode string? Well, this default encoding scheme is an option which you can customize.

Example 9.15. `sitecustomize.py`

```
# sitecustomize.py                           ❶
# this file can be anywhere in your Python path,
# but it usually goes in ${pythondir}/lib/site−packages/
import sys
sys.setdefaultencoding('iso−8859−1') ❷
```

1. `sitecustomize.py` is a special script; Python will try to import it on startup, so any code in it will be run automatically. As the comment mentions, it can go anywhere (as long as `import` can find it), but it usually goes in the `site−packages` directory within your Python `lib` directory.
2. `setdefaultencoding` function sets, well, the default encoding. This is the encoding scheme that Python will try to use whenever it needs to auto−coerce a unicode string into a regular string.

Example 9.16. Effects of setting the default encoding

```
>>> import sys
>>> sys.getdefaultencoding() ❶
'iso−8859−1'
>>> s = u'La Pe\xf1a'
>>> print s                  ❷
La Peña
```

1. This example assumes that you have made the changes listed in the previous example to your `sitecustomize.py` file, and restarted Python. If your default encoding still says `'ascii'`, you didn't set up your `sitecustomize.py` properly, or you didn't restart Python. The default encoding can only be changed during Python startup; you can't change it later. (Due to some wacky programming tricks that I won't get into right now, you can't even call `sys.setdefaultencoding` after Python has started up. Dig into `site.py` and search for "`setdefaultencoding`" to find out how.)
2. Now that the default encoding scheme includes all the characters you use in your string, Python has no problem auto−coercing the string and printing it.

Example 9.17. Specifying encoding in `.py` files

If you are going to be storing non−ASCII strings within your Python code, you'll need to specify the encoding of each individual `.py` file by putting an encoding declaration at the top of each file. This declaration defines the `.py` file to be UTF−8:

```
#!/usr/bin/env python
# −*− coding: UTF−8 −*−
```

Now, what about XML? Well, every XML document is in a specific encoding. Again, ISO−8859−1 is a popular encoding for data in Western European languages. KOI8−R is popular for Russian texts. The encoding, if specified, is in the header of the XML document.

Example 9.18. `russiansample.xml`

```
<?xml version="1.0" encoding="koi8-r"?>         ❶
<preface>
<title> @548A;>285</title>                      ❷
</preface>
```

❶ This is a sample extract from a real Russian XML document; it's part of a Russian translation of this very book. Note the encoding, `koi8−r`, specified in the header.

❷ These are Cyrillic characters which, as far as I know, spell the Russian word for "Preface". If you open this file in a regular text editor, the characters will most likely like gibberish, because they're encoded using the `koi8−r` encoding scheme, but they're being displayed in `iso−8859−1`.

Example 9.19. Parsing `russiansample.xml`

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('russiansample.xml') ❶
>>> title = xmldoc.getElementsByTagName('title')[0].firstChild.data
>>> title                                        ❷
u'\u041f\u0440\u0435\u0434\u0438\u0441\u043b\u043e\u0432\u0438\u0435'
>>> print title                                  ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> convertedtitle = title.encode('koi8-r')      ❹
>>> convertedtitle
'\xf0\xd2\xc5\xc4\xc9\xd3\xcc\xcf\xd7\xc9\xc5'
>>> print convertedtitle                         ❺
 @548A;>285
```

❶ I'm assuming here that you saved the previous example as `russiansample.xml` in the current directory. I am also, for the sake of completeness, assuming that you've changed your default encoding back to `'ascii'` by removing your `sitecustomize.py` file, or at least commenting out the `setdefaultencoding` line.

❷ Note that the text data of the `title` tag (now in the `title` variable, thanks to that long concatenation of Python functions which I hastily skipped over and, annoyingly, won't explain until the next section) −− the text data inside the XML document's `title` element is stored in unicode.

❸ Printing the title is not possible, because this unicode string contains non−ASCII characters, so Python can't convert it to ASCII because that doesn't make sense.

❹ You can, however, explicitly convert it to `koi8−r`, in which case you get a (regular, not unicode) string of single−byte characters (`f0`, `d2`, `c5`, and so forth) that are the `koi8−r`−encoded versions

of the characters in the original unicode string.

❺ Printing the `koi8-r`-encoded string will probably show gibberish on your screen, because your Python IDE is interpreting those characters as `iso-8859-1`, not `koi8-r`. But at least they do print. (And, if you look carefully, it's the same gibberish that you saw when you opened the original XML document in a non-unicode-aware text editor. Python converted it from `koi8-r` into unicode when it parsed the XML document, and you've just converted it back.)

To sum up, unicode itself is a bit intimidating if you've never seen it before, but unicode data is really very easy to handle in Python. If your XML documents are all 7-bit ASCII (like the examples in this chapter), you will literally never think about unicode. Python will convert the ASCII data in the XML documents into unicode while parsing, and auto-coerce it back to ASCII whenever necessary, and you'll never even notice. But if you need to deal with that in other languages, Python is ready.

Further reading

- Unicode.org (http://www.unicode.org/) is the home page of the unicode standard, including a brief technical introduction (http://www.unicode.org/standard/principles.html).
- Unicode Tutorial (http://www.reportlab.com/i18n/python_unicode_tutorial.html) has some more examples of how to use Python's unicode functions, including how to force Python to coerce unicode into ASCII even when it doesn't really want to.
- PEP 263 (http://www.python.org/peps/pep-0263.html) goes into more detail about how and when to define a character encoding in your `.py` files.

9.5. Searching for elements

Traversing XML documents by stepping through each node can be tedious. If you're looking for something in particular, buried deep within your XML document, there is a shortcut you can use to find it quickly: `getElementsByTagName`.

For this section, you'll be using the `binary.xml` grammar file, which looks like this:

Example 9.20. `binary.xml`

```
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "-//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

It has two `ref`s, `'bit'` and `'byte'`. A `bit` is either a `'0'` or `'1'`, and a `byte` is 8 `bit`s.

Example 9.21. Introducing `getElementsByTagName`

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref') ❶
```

```
>>> reflist
[<DOM Element: ref at 136138108>, <DOM Element: ref at 136144292>]
>>> print reflist[0].toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print reflist[1].toxml()
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
```

❶ `getElementsByTagName` takes one argument, the name of the element you wish to find. It returns a list of `Element` objects, corresponding to the XML elements that have that name. In this case, you find two `ref` elements.

Example 9.22. Every element is searchable

```
>>> firstref = reflist[0]                     ❶
>>> print firstref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> plist = firstref.getElementsByTagName("p") ❷
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>]
>>> print plist[0].toxml()                    ❸
<p>0</p>
>>> print plist[1].toxml()
<p>1</p>
```

❶ Continuing from the previous example, the first object in your `reflist` is the `'bit'` `ref` element.

❷ You can use the same `getElementsByTagName` method on this `Element` to find all the `<p>` elements within the `'bit'` `ref` element.

❸ Just as before, the `getElementsByTagName` method returns a list of all the elements it found. In this case, you have two, one for each bit.

Example 9.23. Searching is actually recursive

```
>>> plist = xmldoc.getElementsByTagName("p") ❶
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>, <DOM Element: p at 136146124>]
>>> plist[0].toxml()                         ❷
'<p>0</p>'
>>> plist[1].toxml()
'<p>1</p>'
>>> plist[2].toxml()                         ❸
'<p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>'
```

❶ Note carefully the difference between this and the previous example. Previously, you were searching for `p` elements within `firstref`, but here you are searching for `p` elements within `xmldoc`, the root−level object that represents the entire XML document. This *does* find the `p` elements nested within the `ref` elements within the root `grammar` element.

❷ The first two `p` elements are within the first `ref` (the `'bit'` `ref`).

❸ The last `p` element is the one within the second `ref` (the `'byte' ref`).

9.6. Accessing element attributes

XML elements can have one or more attributes, and it is incredibly simple to access them once you have parsed an XML document.

For this section, you'll be using the `binary.xml` grammar file that you saw in the previous section.

This section may be a little confusing, because of some overlapping terminology. Elements in an XML document have attributes, and Python objects also have attributes. When you parse an XML document, you get a bunch of Python objects that represent all the pieces of the XML document, and some of these Python objects represent attributes of the XML elements. But the (Python) objects that represent the (XML) attributes also have (Python) attributes, which are used to access various parts of the (XML) attribute that the object represents. I told you it was confusing. I am open to suggestions on how to distinguish these more clearly.

Example 9.24. Accessing element attributes

```
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref')
>>> bitref = reflist[0]
>>> print bitref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> bitref.attributes          ❶
<xml.dom.minidom.NamedNodeMap instance at 0x81e0c9c>
>>> bitref.attributes.keys()   ❷ ❸
[u'id']
>>> bitref.attributes.values() ❹
[<xml.dom.minidom.Attr instance at 0x81d5044>]
>>> bitref.attributes["id"]    ❺
<xml.dom.minidom.Attr instance at 0x81d5044>
```

❶ Each `Element` object has an attribute called `attributes`, which is a `NamedNodeMap` object. This sounds scary, but it's not, because a `NamedNodeMap` is an object that acts like a dictionary, so you already know how to use it.

❷ Treating the `NamedNodeMap` as a dictionary, you can get a list of the names of the attributes of this element by using `attributes.keys()`. This element has only one attribute, `'id'`.

❸ Attribute names, like all other text in an XML document, are stored in unicode.

❹ Again treating the `NamedNodeMap` as a dictionary, you can get a list of the values of the attributes by using `attributes.values()`. The values are themselves objects, of type `Attr`. You'll see how to get useful information out of this object in the next example.

❺ Still treating the `NamedNodeMap` as a dictionary, you can access an individual attribute by name, using normal dictionary syntax. (Readers who have been paying extra−close attention will already know how the `NamedNodeMap` class accomplishes this neat trick: by defining a `__getitem__` special method. Other readers can take comfort in the fact that they don't need to understand how it works in order to use it effectively.)

Example 9.25. Accessing individual attributes

```
>>> a = bitref.attributes["id"]
```

```
>>> a
<xml.dom.minidom.Attr instance at 0x81d5044>
>>> a.name  ❶
u'id'
>>> a.value ❷
u'bit'
```

❶ The `Attr` object completely represents a single XML attribute of a single XML element. The name of the attribute (the same name as you used to find this object in the `bitref.attributes` `NamedNodeMap` pseudo−dictionary) is stored in `a.name`.

❷ The actual text value of this XML attribute is stored in `a.value`.

Like a dictionary, attributes of an XML element have no ordering. Attributes may *happen to be* listed in a certain order in the original XML document, and the `Attr` objects may *happen to be* listed in a certain order when the XML document is parsed into Python objects, but these orders are arbitrary and should carry no special meaning. You should always access individual attributes by name, like the keys of a dictionary.

9.7. Segue

OK, that's it for the hard−core XML stuff. The next chapter will continue to use these same example programs, but focus on other aspects that make the program more flexible: using streams for input processing, using `getattr` for method dispatching, and using command−line flags to allow users to reconfigure the program without changing the code.

Before moving on to the next chapter, you should be comfortable doing all of these things:

- Parsing XML documents using `minidom`, searching through the parsed document, and accessing arbitrary element attributes and element children
- Organizing complex libraries into packages
- Converting unicode strings to different character encodings

---

[5] This, sadly, is *still* an oversimplification. Unicode now has been extended to handle ancient Chinese, Korean, and Japanese texts, which had so many different characters that the 2−byte unicode system could not represent them all. But Python doesn't currently support that out of the box, and I don't know if there is a project afoot to add it. You've reached the limits of my expertise, sorry.

Chapter 10. Scripts and Streams

10.1. Abstracting input sources

One of Python's greatest strengths is its dynamic binding, and one powerful use of dynamic binding is the *file−like object*.

Many functions which require an input source could simply take a filename, go open the file for reading, read it, and close it when they're done. But they don't. Instead, they take a *file−like object*.

In the simplest case, a *file−like object* is any object with a `read` method with an optional `size` parameter, which returns a string. When called with no `size` parameter, it reads everything there is to read from the input source and returns all the data as a single string. When called with a `size` parameter, it reads that much from the input source and returns that much data; when called again, it picks up where it left off and returns the next chunk of data.

This is how reading from real files works; the difference is that you're not limiting yourself to real files. The input source could be anything: a file on disk, a web page, even a hard−coded string. As long as you pass a file−like object to the function, and the function simply calls the object's `read` method, the function can handle any kind of input source without specific code to handle each kind.

In case you were wondering how this relates to XML processing, `minidom.parse` is one such function which can take a file−like object.

Example 10.1. Parsing XML from a file

```
>>> from xml.dom import minidom
>>> fsock = open('binary.xml')    ❶
>>> xmldoc = minidom.parse(fsock) ❷
>>> fsock.close()                 ❸
>>> print xmldoc.toxml()          ❹
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

❶ First, you open the file on disk. This gives you a file object.

❷ You pass the file object to `minidom.parse`, which calls the `read` method of `fsock` and reads the XML document from the file on disk.

❸ Be sure to call the `close` method of the file object after you're done with it. `minidom.parse` will not do this for you.

❹ Calling the `toxml()` method on the returned XML document prints out the entire thing.

Well, that all seems like a colossal waste of time. After all, you've already seen that `minidom.parse` can simply take the filename and do all the opening and closing nonsense automatically. And it's true that if you know you're just going to be parsing a local file, you can pass the filename and `minidom.parse` is smart enough to Do The Right Thing(tm). But notice how similar −− and easy −− it is to parse an XML document straight from the Internet.

Example 10.2. Parsing XML from a URL

```
>>> import urllib
>>> usock = urllib.urlopen('http://slashdot.org/slashdot.rdf') ❶
>>> xmldoc = minidom.parse(usock)                              ❷
>>> usock.close()                                              ❸
>>> print xmldoc.toxml()                                       ❹
<?xml version="1.0" ?>
<rdf:RDF xmlns="http://my.netscape.com/rdf/simple/0.9/"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<channel>
<title>Slashdot</title>
<link>http://slashdot.org/</link>
<description>News for nerds, stuff that matters</description>
</channel>

<image>
<title>Slashdot</title>
<url>http://images.slashdot.org/topics/topicslashdot.gif</url>
<link>http://slashdot.org/</link>
</image>

<item>
<title>To HDTV or Not to HDTV?</title>
<link>http://slashdot.org/article.pl?sid=01/12/28/0421241</link>
</item>

[...snip...]
```

❶ As you saw in a previous chapter, `urlopen` takes a web page URL and returns a file−like object. Most importantly, this object has a `read` method which returns the HTML source of the web page.

❷ Now you pass the file−like object to `minidom.parse`, which obediently calls the `read` method of the object and parses the XML data that the `read` method returns. The fact that this XML data is now coming straight from a web page is completely irrelevant. `minidom.parse` doesn't know about web pages, and it doesn't care about web pages; it just knows about file−like objects.

❸ As soon as you're done with it, be sure to close the file−like object that `urlopen` gives you.

❹ By the way, this URL is real, and it really is XML. It's an XML representation of the current headlines on Slashdot (http://slashdot.org/), a technical news and gossip site.

Example 10.3. Parsing XML from a string (the easy but inflexible way)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> xmldoc = minidom.parseString(contents) ❶
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

❶ `minidom` has a method, `parseString`, which takes an entire XML document as a string and parses it. You can use this instead of `minidom.parse` if you know you already have your entire XML document in a string.

OK, so you can use the `minidom.parse` function for parsing both local files and remote URLs, but for parsing strings, you use... a different function. That means that if you want to be able to take input from a file, a URL, or a string, you'll need special logic to check whether it's a string, and call the `parseString` function instead. How unsatisfying.

If there were a way to turn a string into a file−like object, then you could simply pass this object to `minidom.parse`. And in fact, there is a module specifically designed for doing just that: `StringIO`.

Example 10.4. Introducing `StringIO`

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> import StringIO
>>> ssock = StringIO.StringIO(contents)   ➊
>>> ssock.read()                          ➋
"<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock.read()                          ➌
''
>>> ssock.seek(0)                         ➍
>>> ssock.read(15)                        ➎
'<grammar><ref i'
>>> ssock.read(15)
"d='bit'><p>0</p"
>>> ssock.read()
'><p>1</p></ref></grammar>'
>>> ssock.close()                         ➏
```

➊ The `StringIO` module contains a single class, also called `StringIO`, which allows you to turn a string into a file−like object. The `StringIO` class takes the string as a parameter when creating an instance.

➋ Now you have a file−like object, and you can do all sorts of file−like things with it. Like `read`, which returns the original string.

➌ Calling `read` again returns an empty string. This is how real file objects work too; once you read the entire file, you can't read any more without explicitly seeking to the beginning of the file. The `StringIO` object works the same way.

➍ You can explicitly seek to the beginning of the string, just like seeking through a file, by using the `seek` method of the `StringIO` object.

➎ You can also read the string in chunks, by passing a `size` parameter to the `read` method.

➏ At any time, `read` will return the rest of the string that you haven't read yet. All of this is exactly how file objects work; hence the term *file−like object*.

Example 10.5. Parsing XML from a string (the file−like object way)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock = StringIO.StringIO(contents)
>>> xmldoc = minidom.parse(ssock) ➊
>>> ssock.close()
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

➊ Now you can pass the file−like object (really a `StringIO`) to `minidom.parse`, which will call the object's `read` method and happily parse away, never knowing that its input came from a hard−coded string.

So now you know how to use a single function, `minidom.parse`, to parse an XML document stored on a web page, in a local file, or in a hard−coded string. For a web page, you use `urlopen` to get a file−like object; for a local file, you use `open`; and for a string, you use `StringIO`. Now let's take it one step further and generalize *these* differences as well.

Example 10.6. `openAnything`

```
def openAnything(source):                  ➊
    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
```

```
        return urllib.urlopen(source)      ❷
    except (IOError, OSError):
        pass

    # try to open with native open function (if source is pathname)
    try:
        return open(source)                ❸
    except (IOError, OSError):
        pass

    # treat source as string
    import StringIO
    return StringIO.StringIO(str(source))  ❹
```

❶ The `openAnything` function takes a single parameter, `source`, and returns a file−like object. `source` is a string of some sort; it can either be a URL (like `'http://slashdot.org/slashdot.rdf'`), a full or partial pathname to a local file (like `'binary.xml'`), or a string that contains actual XML data to be parsed.

❷ First, you see if `source` is a URL. You do this through brute force: you try to open it as a URL and silently ignore errors caused by trying to open something which is not a URL. This is actually elegant in the sense that, if `urllib` ever supports new types of URLs in the future, you will also support them without recoding. If `urllib` is able to open `source`, then the `return` kicks you out of the function immediately and the following `try` statements never execute.

❸ On the other hand, if `urllib` yelled at you and told you that `source` wasn't a valid URL, you assume it's a path to a file on disk and try to open it. Again, you don't do anything fancy to check whether `source` is a valid filename or not (the rules for valid filenames vary wildly between different platforms anyway, so you'd probably get them wrong anyway). Instead, you just blindly open the file, and silently trap any errors.

❹ By this point, you need to assume that `source` is a string that has hard−coded data in it (since nothing else worked), so you use `StringIO` to create a file−like object out of it and return that. (In fact, since you're using the `str` function, `source` doesn't even need to be a string; it could be any object, and you'll use its string representation, as defined by its `__str__` special method.)

Now you can use this `openAnything` function in conjunction with `minidom.parse` to make a function that takes a `source` that refers to an XML document somehow (either as a URL, or a local filename, or a hard−coded XML document in a string) and parses it.

Example 10.7. Using `openAnything` in `kgp.py`

```
class KantGenerator:
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc
```

10.2. Standard input, output, and error

UNIX users are already familiar with the concept of standard input, standard output, and standard error. This section is for the rest of you.

Standard output and standard error (commonly abbreviated `stdout` and `stderr`) are pipes that are built into every UNIX system. When you `print` something, it goes to the `stdout` pipe; when your program crashes and prints out debugging information (like a traceback in Python), it goes to the `stderr` pipe. Both of these pipes are ordinarily just connected to the terminal window where you are working, so when a program prints, you see the output, and when a program crashes, you see the debugging information. (If you're working on a system with a window−based Python

IDE, `stdout` and `stderr` default to your "Interactive Window".)

Example 10.8. Introducing stdout and stderr

```
>>> for i in range(3):
...     print 'Dive in'             ❶
Dive in
Dive in
Dive in
>>> import sys
>>> for i in range(3):
...     sys.stdout.write('Dive in') ❷
Dive inDive inDive in
>>> for i in range(3):
...     sys.stderr.write('Dive in') ❸
Dive inDive inDive in
```

❶ As you saw in Example 6.9, Simple Counters , you can use Python's built−in `range` function to build simple counter loops that repeat something a set number of times.

❷ `stdout` is a file−like object; calling its `write` function will print out whatever string you give it. In fact, this is what the `print` function really does; it adds a carriage return to the end of the string you're printing, and calls `sys.stdout.write`.

❸ In the simplest case, `stdout` and `stderr` send their output to the same place: the Python IDE (if you're in one), or the terminal (if you're running Python from the command line). Like `stdout`, `stderr` does not add carriage returns for you; if you want them, add them yourself.

`stdout` and `stderr` are both file−like objects, like the ones you discussed in Section 10.1, Abstracting input sources , but they are both write−only. They have no `read` method, only `write`. Still, they are file−like objects, and you can assign any other file− or file−like object to them to redirect their output.

Example 10.9. Redirecting output

```
[you@localhost kgp]$ python stdout.py
Dive in
[you@localhost kgp]$ cat out.log
This message will be logged instead of displayed
```

(On Windows, you can use `type` instead of `cat` to display the contents of a file.)

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
#stdout.py
import sys

print 'Dive in'                                          ❶
saveout = sys.stdout                                     ❷
fsock = open('out.log', 'w')                             ❸
sys.stdout = fsock                                       ❹
print 'This message will be logged instead of displayed' ❺
sys.stdout = saveout                                     ❻
fsock.close()                                            ❼
```

❶ This will print to the IDE "Interactive Window" (or the terminal, if running the script from the command line).

❷ Always save `stdout` before redirecting it, so you can set it back to normal later.
❸ Open a file for writing. If the file doesn't exist, it will be created. If the file does exist, it will be overwritten.
❹ Redirect all further output to the new file you just opened.
❺ This will be "printed" to the log file only; it will not be visible in the IDE window or on the screen.
❻ Set `stdout` back to the way it was before you mucked with it.
❼ Close the log file.

Redirecting `stderr` works exactly the same way, using `sys.stderr` instead of `sys.stdout`.

Example 10.10. Redirecting error information

```
[you@localhost kgp]$ python stderr.py
[you@localhost kgp]$ cat error.log
Traceback (most recent line last):
  File "stderr.py", line 5, in ?
    raise Exception, 'this error will be logged'
Exception: this error will be logged
```

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
#stderr.py
import sys

fsock = open('error.log', 'w')               ❶
sys.stderr = fsock                           ❷
raise Exception, 'this error will be logged' ❸ ❹
```

❶ Open the log file where you want to store debugging information.
❷ Redirect standard error by assigning the file object of the newly−opened log file to `stderr`.
❸ Raise an exception. Note from the screen output that this does *not* print anything on screen. All the normal traceback information has been written to `error.log`.
❹ Also note that you're not explicitly closing your log file, nor are you setting `stderr` back to its original value. This is fine, since once the program crashes (because of the exception), Python will clean up and close the file for us, and it doesn't make any difference that `stderr` is never restored, since, as I mentioned, the program crashes and Python ends. Restoring the original is more important for `stdout`, if you expect to go do other stuff within the same script afterwards.

Since it is so common to write error messages to standard error, there is a shorthand syntax that can be used instead of going through the hassle of redirecting it outright.

Example 10.11. Printing to `stderr`

```
>>> print 'entering function'
entering function
>>> import sys
>>> print >> sys.stderr, 'entering function' ❶
entering function
```

❶ This shorthand syntax of the `print` statement can be used to write to any open file, or file−like object. In this case, you can redirect a single `print` statement to `stderr` without affecting subsequent `print` statements.

Standard input, on the other hand, is a read−only file object, and it represents the data flowing into the program from some previous program. This will likely not make much sense to classic Mac OS users, or even Windows users unless you were ever fluent on the MS−DOS command line. The way it works is that you can construct a chain of commands in a single line, so that one program's output becomes the input for the next program in the chain. The first program simply outputs to standard output (without doing any special redirecting itself, just doing normal `print` statements or whatever), and the next program reads from standard input, and the operating system takes care of connecting one program's output to the next program's input.

Example 10.12. Chaining commands

```
[you@localhost kgp]$ python kgp.py −g binary.xml         ❶
01100111
[you@localhost kgp]$ cat binary.xml                       ❷
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "−//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
[you@localhost kgp]$ cat binary.xml | python kgp.py −g − ❸ ❹
10110001
```

❶ As you saw in Section 9.1, Diving in , this will print a string of eight random bits, `0` or `1`.

❷ This simply prints out the entire contents of `binary.xml`. (Windows users should use `type` instead of `cat`.)

❸ This prints the contents of `binary.xml`, but the "`|`" character, called the "pipe" character, means that the contents will not be printed to the screen. Instead, they will become the standard input of the next command, which in this case calls your Python script.

❹ Instead of specifying a module (like `binary.xml`), you specify "`−`", which causes your script to load the grammar from standard input instead of from a specific file on disk. (More on how this happens in the next example.) So the effect is the same as the first syntax, where you specified the grammar filename directly, but think of the expansion possibilities here. Instead of simply doing `cat binary.xml`, you could run a script that dynamically generates the grammar, then you can pipe it into your script. It could come from anywhere: a database, or some grammar−generating meta−script, or whatever. The point is that you don't need to change your `kgp.py` script at all to incorporate any of this functionality. All you need to do is be able to take grammar files from standard input, and you can separate all the other logic into another program.

So how does the script "know" to read from standard input when the grammar file is "`−`"? It's not magic; it's just code.

Example 10.13. Reading from standard input in `kgp.py`

```
def openAnything(source):
    if source == "−":     ❶
        import sys
        return sys.stdin

    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
```

```
[... snip ...]
```

❶ This is the openAnything function from toolbox.py, which you previously examined in Section 10.1, Abstracting input sources . All you've done is add three lines of code at the beginning of the function to check if the source is "−"; if so, you return sys.stdin. Really, that's it! Remember, stdin is a file−like object with a read method, so the rest of the code (in kgp.py, where you call openAnything) doesn't change a bit.

10.3. Caching node lookups

kgp.py employs several tricks which may or may not be useful to you in your XML processing. The first one takes advantage of the consistent structure of the input documents to build a cache of nodes.

A grammar file defines a series of ref elements. Each ref contains one or more p elements, which can contain a lot of different things, including xrefs. Whenever you encounter an xref, you look for a corresponding ref element with the same id attribute, and choose one of the ref element's children and parse it. (You'll see how this random choice is made in the next section.)

This is how you build up the grammar: define ref elements for the smallest pieces, then define ref elements which "include" the first ref elements by using xref, and so forth. Then you parse the "largest" reference and follow each xref, and eventually output real text. The text you output depends on the (random) decisions you make each time you fill in an xref, so the output is different each time.

This is all very flexible, but there is one downside: performance. When you find an xref and need to find the corresponding ref element, you have a problem. The xref has an id attribute, and you want to find the ref element that has that same id attribute, but there is no easy way to do that. The slow way to do it would be to get the entire list of ref elements each time, then manually loop through and look at each id attribute. The fast way is to do that once and build a cache, in the form of a dictionary.

Example 10.14. loadGrammar

```
    def loadGrammar(self, grammar):
        self.grammar = self._load(grammar)
        self.refs = {}                                           ❶
        for ref in self.grammar.getElementsByTagName("ref"): ❷
            self.refs[ref.attributes["id"].value] = ref          ❸ ❹
```

❶ Start by creating an empty dictionary, self.refs.

❷ As you saw in Section 9.5, Searching for elements , getElementsByTagName returns a list of all the elements of a particular name. You easily can get a list of all the ref elements, then simply loop through that list.

❸ As you saw in Section 9.6, Accessing element attributes , you can access individual attributes of an element by name, using standard dictionary syntax. So the keys of the self.refs dictionary will be the values of the id attribute of each ref element.

❹ The values of the self.refs dictionary will be the ref elements themselves. As you saw in Section 9.3, Parsing XML , each element, each node, each comment, each piece of text in a parsed XML document is an object.

Once you build this cache, whenever you come across an xref and need to find the ref element with the same id attribute, you can simply look it up in self.refs.

Example 10.15. Using the ref element cache

```
    def do_xref(self, node):
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
```

You'll explore the `randomChildElement` function in the next section.

10.4. Finding direct children of a node

Another useful techique when parsing XML documents is finding all the direct child elements of a particular element. For instance, in the grammar files, a `ref` element can have several `p` elements, each of which can contain many things, including other `p` elements. You want to find just the `p` elements that are children of the `ref`, not `p` elements that are children of other `p` elements.

You might think you could simply use `getElementsByTagName` for this, but you can't. `getElementsByTagName` searches recursively and returns a single list for all the elements it finds. Since `p` elements can contain other `p` elements, you can't use `getElementsByTagName`, because it would return nested `p` elements that you don't want. To find only direct child elements, you'll need to do it yourself.

Example 10.16. Finding direct child elements

```
    def randomChildElement(self, node):
        choices = [e for e in node.childNodes
                   if e.nodeType == e.ELEMENT_NODE] ❶ ❷ ❸
        chosen = random.choice(choices)             ❹
        return chosen
```

❶ As you saw in Example 9.9, Getting child nodes , the `childNodes` attribute returns a list of all the child nodes of an element.

❷ However, as you saw in Example 9.11, Child nodes can be text , the list returned by `childNodes` contains all different types of nodes, including text nodes. That's not what you're looking for here. You only want the children that are elements.

❸ Each node has a `nodeType` attribute, which can be `ELEMENT_NODE`, `TEXT_NODE`, `COMMENT_NODE`, or any number of other values. The complete list of possible values is in the `__init__.py` file in the `xml.dom` package. (See Section 9.2, Packages for more on packages.) But you're just interested in nodes that are elements, so you can filter the list to only include those nodes whose `nodeType` is `ELEMENT_NODE`.

❹ Once you have a list of actual elements, choosing a random one is easy. Python comes with a module called `random` which includes several useful functions. The `random.choice` function takes a list of any number of items and returns a random item. For example, if the `ref` elements contains several `p` elements, then `choices` would be a list of `p` elements, and `chosen` would end up being assigned exactly one of them, selected at random.

10.5. Creating separate handlers by node type

The third useful XML processing tip involves separating your code into logical functions, based on node types and element names. Parsed XML documents are made up of various types of nodes, each represented by a Python object. The root level of the document itself is represented by a `Document` object. The `Document` then contains one or more `Element` objects (for actual XML tags), each of which may contain other `Element` objects, `Text` objects (for bits of text), or `Comment` objects (for embedded comments). Python makes it easy to write a dispatcher to separate the logic for each node type.

Example 10.17. Class names of parsed XML objects

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('kant.xml') ❶
>>> xmldoc
<xml.dom.minidom.Document instance at 0x01359DE8>
>>> xmldoc.__class__                  ❷
<class xml.dom.minidom.Document at 0x01105D40>
>>> xmldoc.__class__.__name__         ❸
'Document'
```

❶ Assume for a moment that `kant.xml` is in the current directory.

❷ As you saw in Section 9.2, Packages , the object returned by parsing an XML document is a `Document` object, as defined in the `minidom.py` in the `xml.dom` package. As you saw in Section 5.4, Instantiating Classes , `__class__` is built−in attribute of every Python object.

❸ Furthermore, `__name__` is a built−in attribute of every Python class, and it is a string. This string is not mysterious; it's the same as the class name you type when you define a class yourself. (See Section 5.3, Defining Classes .)

Fine, so now you can get the class name of any particular XML node (since each XML node is represented as a Python object). How can you use this to your advantage to separate the logic of parsing each node type? The answer is `getattr`, which you first saw in Section 4.4, Getting Object References With getattr .

Example 10.18. `parse`, a generic XML node dispatcher

```
    def parse(self, node):
        parseMethod = getattr(self, "parse_%s" % node.__class__.__name__) ❶ ❷
        parseMethod(node) ❸
```

❶ First off, notice that you're constructing a larger string based on the class name of the node you were passed (in the `node` argument). So if you're passed a `Document` node, you're constructing the string `'parse_Document'`, and so forth.

❷ Now you can treat that string as a function name, and get a reference to the function itself using `getattr`

❸ Finally, you can call that function and pass the node itself as an argument. The next example shows the definitions of each of these functions.

Example 10.19. Functions called by the `parse` dispatcher

```
    def parse_Document(self, node): ❶
        self.parse(node.documentElement)

    def parse_Text(self, node):    ❷
        text = node.data
        if self.capitalizeNextWord:
            self.pieces.append(text[0].upper())
            self.pieces.append(text[1:])
            self.capitalizeNextWord = 0
        else:
            self.pieces.append(text)

    def parse_Comment(self, node): ❸
        pass

    def parse_Element(self, node): ❹
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)
```

1. `parse_Document` is only ever called once, since there is only one `Document` node in an XML document, and only one `Document` object in the parsed XML representation. It simply turns around and parses the root element of the grammar file.
2. `parse_Text` is called on nodes that represent bits of text. The function itself does some special processing to handle automatic capitalization of the first word of a sentence, but otherwise simply appends the represented text to a list.
3. `parse_Comment` is just a `pass`, since you don't care about embedded comments in the grammar files. Note, however, that you still need to define the function and explicitly make it do nothing. If the function did not exist, the generic `parse` function would fail as soon as it stumbled on a comment, because it would try to find the non−existent `parse_Comment` function. Defining a separate function for every node type, even ones you don't use, allows the generic `parse` function to stay simple and dumb.
4. The `parse_Element` method is actually itself a dispatcher, based on the name of the element's tag. The basic idea is the same: take what distinguishes elements from each other (their tag names) and dispatch to a separate function for each of them. You construct a string like `'do_xref'` (for an `<xref>` tag), find a function of that name, and call it. And so forth for each of the other tag names that might be found in the course of parsing a grammar file (`<p>` tags, `<choice>` tags).

In this example, the dispatch functions `parse` and `parse_Element` simply find other methods in the same class. If your processing is very complex (or you have many different tag names), you could break up your code into separate modules, and use dynamic importing to import each module and call whatever functions you needed. Dynamic importing will be discussed in Chapter 16, *Functional Programming*.

10.6. Handling command−line arguments

Python fully supports creating programs that can be run on the command line, complete with command−line arguments and either short− or long−style flags to specify various options. None of this is XML−specific, but this script makes good use of command−line processing, so it seemed like a good time to mention it.

It's difficult to talk about command−line processing without understanding how command−line arguments are exposed to your Python program, so let's write a simple program to see them.

Example 10.20. Introducing `sys.argv`

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
#argecho.py
import sys

for arg in sys.argv: ❶
    print arg
```

1. Each command−line argument passed to the program will be in `sys.argv`, which is just a list. Here you are printing each argument on a separate line.

Example 10.21. The contents of `sys.argv`

```
[you@localhost py]$ python argecho.py             ❶
argecho.py
[you@localhost py]$ python argecho.py abc def     ❷
argecho.py
abc
def
```

```
[you@localhost py]$ python argecho.py --help      ❸
argecho.py
--help
[you@localhost py]$ python argecho.py -m kant.xml ❹
argecho.py
-m
kant.xml
```

❶ The first thing to know about `sys.argv` is that it contains the name of the script you're calling. You will actually use this knowledge to your advantage later, in Chapter 16, *Functional Programming*. Don't worry about it for now.

❷ Command−line arguments are separated by spaces, and each shows up as a separate element in the `sys.argv` list.

❸ Command−line flags, like `--help`, also show up as their own element in the `sys.argv` list.

❹ To make things even more interesting, some command−line flags themselves take arguments. For instance, here you have a flag (`-m`) which takes an argument (`kant.xml`). Both the flag itself and the flag's argument are simply sequential elements in the `sys.argv` list. No attempt is made to associate one with the other; all you get is a list.

So as you can see, you certainly have all the information passed on the command line, but then again, it doesn't look like it's going to be all that easy to actually use it. For simple programs that only take a single argument and have no flags, you can simply use `sys.argv[1]` to access the argument. There's no shame in this; I do it all the time. For more complex programs, you need the `getopt` module.

Example 10.22. Introducing `getopt`

```
def main(argv):
    grammar = "kant.xml"                 ❶
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="]) ❷
    except getopt.GetoptError:           ❸
        usage()                          ❹
        sys.exit(2)

...

if __name__ == "__main__":
    main(sys.argv[1:])
```

❶ First off, look at the bottom of the example and notice that you're calling the `main` function with `sys.argv[1:]`. Remember, `sys.argv[0]` is the name of the script that you're running; you don't care about that for command−line processing, so you chop it off and pass the rest of the list.

❷ This is where all the interesting processing happens. The `getopt` function of the `getopt` module takes three parameters: the argument list (which you got from `sys.argv[1:]`), a string containing all the possible single−character command−line flags that this program accepts, and a list of longer command−line flags that are equivalent to the single−character versions. This is quite confusing at first glance, and is explained in more detail below.

❸ If anything goes wrong trying to parse these command−line flags, `getopt` will raise an exception, which you catch. You told `getopt` all the flags you understand, so this probably means that the end user passed some command−line flag that you don't understand.

❹ As is standard practice in the UNIX world, when the script is passed flags it doesn't understand, you print out a summary of proper usage and exit gracefully. Note that I haven't shown the `usage` function here. You would still need to code that somewhere and have it print out the appropriate summary; it's not automatic.

So what are all those parameters you pass to the `getopt` function? Well, the first one is simply the raw list of command−line flags and arguments (not including the first element, the script name, which you already chopped off before calling the `main` function). The second is the list of short command−line flags that the script accepts.

`"hg:d"`

`−h`
: print usage summary

`−g ...`
: use specified grammar file or URL

`−d`
: show debugging information while parsing

The first and third flags are simply standalone flags; you specify them or you don't, and they do things (print help) or change state (turn on debugging). However, the second flag (`−g`) *must* be followed by an argument, which is the name of the grammar file to read from. In fact it can be a filename or a web address, and you don't know which yet (you'll figure it out later), but you know it has to be *something*. So you tell `getopt` this by putting a colon after the `g` in that second parameter to the `getopt` function.

To further complicate things, the script accepts either short flags (like `−h`) or long flags (like `−−help`), and you want them to do the same thing. This is what the third parameter to `getopt` is for, to specify a list of the long flags that correspond to the short flags you specified in the second parameter.

`["help", "grammar="]`

`−−help`
: print usage summary

`−−grammar ...`
: use specified grammar file or URL

Three things of note here:

1. All long flags are preceded by two dashes on the command line, but you don't include those dashes when calling `getopt`. They are understood.
2. The `−−grammar` flag must always be followed by an additional argument, just like the `−g` flag. This is notated by an equals sign, `"grammar="`.
3. The list of long flags is shorter than the list of short flags, because the `−d` flag does not have a corresponding long version. This is fine; only `−d` will turn on debugging. But the order of short and long flags needs to be the same, so you'll need to specify all the short flags that *do* have corresponding long flags first, then all the rest of the short flags.

Confused yet? Let's look at the actual code and see if it makes sense in context.

Example 10.23. Handling command−line arguments in `kgp.py`

```
def main(argv):                          ❶
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:                ❷
```

```
        if opt in ("-h", "--help"):      ❸
            usage()
            sys.exit()
        elif opt == '-d':                ❹
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"): ❺
            grammar = arg

    source = "".join(args)               ❻

    k = KantGenerator(grammar, source)
    print k.output()
```

❶ The `grammar` variable will keep track of the grammar file you're using. You initialize it here in case it's not specified on the command line (using either the `-g` or the `--grammar` flag).

❷ The `opts` variable that you get back from `getopt` contains a list of tuples: `flag` and `argument`. If the flag doesn't take an argument, then `arg` will simply be `None`. This makes it easier to loop through the flags.

❸ `getopt` validates that the command−line flags are acceptable, but it doesn't do any sort of conversion between short and long flags. If you specify the `-h` flag, `opt` will contain `"-h"`; if you specify the `--help` flag, `opt` will contain `"--help"`. So you need to check for both.

❹ Remember, the `-d` flag didn't have a corresponding long flag, so you only need to check for the short form. If you find it, you set a global variable that you'll refer to later to print out debugging information. (I used this during the development of the script. What, you thought all these examples worked on the first try?)

❺ If you find a grammar file, either with a `-g` flag or a `--grammar` flag, you save the argument that followed it (stored in `arg`) into the `grammar` variable, overwriting the default that you initialized at the top of the `main` function.

❻ That's it. You've looped through and dealt with all the command−line flags. That means that anything left must be command−line arguments. These come back from the `getopt` function in the `args` variable. In this case, you're treating them as source material for the parser. If there are no command−line arguments specified, `args` will be an empty list, and `source` will end up as the empty string.

10.7. Putting it all together

You've covered a lot of ground. Let's step back and see how all the pieces fit together.

To start with, this is a script that takes its arguments on the command line, using the `getopt` module.

```
def main(argv):
...
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
...
    for opt, arg in opts:
...
```

You create a new instance of the `KantGenerator` class, and pass it the grammar file and source that may or may not have been specified on the command line.

```
    k = KantGenerator(grammar, source)
```

The `KantGenerator` instance automatically loads the grammar, which is an XML file. You use your custom `openAnything` function to open the file (which could be stored in a local file or a remote web server), then use the built−in `minidom` parsing functions to parse the XML into a tree of Python objects.

```
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
```

Oh, and along the way, you take advantage of your knowledge of the structure of the XML document to set up a little cache of references, which are just elements in the XML document.

```
    def loadGrammar(self, grammar):
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref
```

If you specified some source material on the command line, you use that; otherwise you rip through the grammar looking for the "top−level" reference (that isn't referenced by anything else) and use that as a starting point.

```
    def getDefaultSource(self):
        xrefs = {}
        for xref in self.grammar.getElementsByTagName("xref"):
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)
```

Now you rip through the source material. The source material is also XML, and you parse it one node at a time. To keep the code separated and more maintainable, you use separate handlers for each node type.

```
    def parse_Element(self, node):
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)
```

You bounce through the grammar, parsing all the children of each `p` element,

```
    def do_p(self, node):
...
        if doit:
            for child in node.childNodes: self.parse(child)
```

replacing `choice` elements with a random child,

```
    def do_choice(self, node):
        self.parse(self.randomChildElement(node))
```

and replacing `xref` elements with a random child of the corresponding `ref` element, which you previously cached.

```
    def do_xref(self, node):
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
```

Eventually, you parse your way down to plain text,

```
    def parse_Text(self, node):
        text = node.data
...
            self.pieces.append(text)
```

which you print out.

```
def main(argv):
```

```
...
    k = KantGenerator(grammar, source)
    print k.output()
```

10.8. Summary

Python comes with powerful libraries for parsing and manipulating XML documents. The `minidom` takes an XML file and parses it into Python objects, providing for random access to arbitrary elements. Furthermore, this chapter shows how Python can be used to create a "real" standalone command−line script, complete with command−line flags, command−line arguments, error handling, even the ability to take input from the piped result of a previous program.

Before moving on to the next chapter, you should be comfortable doing all of these things:

- Chaining programs with standard input and output
- Defining dynamic dispatchers with `getattr`.
- Using command−line flags and validating them with `getopt`

Chapter 11. HTTP Web Services

11.1. Diving in

You've learned about HTML processing and XML processing, and along the way you saw how to download a web page and how to parse XML from a URL, but let's dive into the more general topic of HTTP web services.

Simply stated, HTTP web services are programmatic ways of sending and receiving data from remote servers using the operations of HTTP directly. If you want to get data from the server, use a straight HTTP GET; if you want to send new data to the server, use HTTP POST. (Some more advanced HTTP web service APIs also define ways of modifying existing data and deleting data, using HTTP PUT and HTTP DELETE.) In other words, the "verbs" built into the HTTP protocol (GET, POST, PUT, and DELETE) map directly to application−level operations for receiving, sending, modifying, and deleting data.

The main advantage of this approach is simplicity, and its simplicity has proven popular with a lot of different sites. Data −− usually XML data −− can be built and stored statically, or generated dynamically by a server−side script, and all major languages include an HTTP library for downloading it. Debugging is also easier, because you can load up the web service in any web browser and see the raw data. Modern browsers will even nicely format and pretty−print XML data for you, to allow you to quickly navigate through it.

Examples of pure XML−over−HTTP web services:

- Amazon API (http://www.amazon.com/webservices) allows you to retrieve product information from the Amazon.com online store.
- National Weather Service (http://www.nws.noaa.gov/alerts/) (United States) and Hong Kong Observatory (http://demo.xml.weather.gov.hk/) (Hong Kong) offer weather alerts as a web service.
- Atom API (http://atomenabled.org/) for managing web−based content.
- Syndicated feeds (http://syndic8.com/) from weblogs and news sites bring you up−to−the−minute news from a variety of sites.

In later chapters, you'll explore APIs which use HTTP as a transport for sending and receiving data, but don't map application semantics to the underlying HTTP semantics. (They tunnel everything over HTTP POST.) But this chapter will concentrate on using HTTP GET to get data from a remote server, and you'll explore several HTTP features you can use to get the maximum benefit out of pure HTTP web services.

Here is a more advanced version of the `openanything` module that you saw in the previous chapter:

Example 11.1. `openanything.py`

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
import urllib2, urlparse, gzip
from StringIO import StringIO

USER_AGENT = 'OpenAnything/1.0 +http://diveintopython.org/http_web_services/'

class SmartRedirectHandler(urllib2.HTTPRedirectHandler):
    def http_error_301(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_301(
            self, req, fp, code, msg, headers)
        result.status = code
```

```
        return result

    def http_error_302(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_302(
            self, req, fp, code, msg, headers)
        result.status = code
        return result

class DefaultErrorHandler(urllib2.HTTPDefaultErrorHandler):
    def http_error_default(self, req, fp, code, msg, headers):
        result = urllib2.HTTPError(
            req.get_full_url(), code, msg, headers, fp)
        result.status = code
        return result

def openAnything(source, etag=None, lastmodified=None, agent=USER_AGENT):
    '''URL, filename, or string --> stream

    This function lets you define parsers that take any input source
    (URL, pathname to local or network file, or actual data as a string)
    and deal with it in a uniform manner.  Returned object is guaranteed
    to have all the basic stdio read methods (read, readline, readlines).
    Just .close() the object when you're done with it.

    If the etag argument is supplied, it will be used as the value of an
    If-None-Match request header.

    If the lastmodified argument is supplied, it must be a formatted
    date/time string in GMT (as returned in the Last-Modified header of
    a previous request).  The formatted date/time will be used
    as the value of an If-Modified-Since request header.

    If the agent argument is supplied, it will be used as the value of a
    User-Agent request header.
    '''

    if hasattr(source, 'read'):
        return source

    if source == '-':
        return sys.stdin

    if urlparse.urlparse(source)[0] == 'http':
        # open URL with urllib2
        request = urllib2.Request(source)
        request.add_header('User-Agent', agent)
        if etag:
            request.add_header('If-None-Match', etag)
        if lastmodified:
            request.add_header('If-Modified-Since', lastmodified)
        request.add_header('Accept-encoding', 'gzip')
        opener = urllib2.build_opener(SmartRedirectHandler(), DefaultErrorHandler())
        return opener.open(request)

    # try to open with native open function (if source is a filename)
    try:
        return open(source)
    except (IOError, OSError):
        pass

    # treat source as string
    return StringIO(str(source))
```

```
def fetch(source, etag=None, last_modified=None, agent=USER_AGENT):
    '''Fetch data and metadata from a URL, file, stream, or string'''
    result = {}
    f = openAnything(source, etag, last_modified, agent)
    result['data'] = f.read()
    if hasattr(f, 'headers'):
        # save ETag, if the server sent one
        result['etag'] = f.headers.get('ETag')
        # save Last-Modified header, if the server sent one
        result['lastmodified'] = f.headers.get('Last-Modified')
        if f.headers.get('content-encoding', '') == 'gzip':
            # data came back gzip-compressed, decompress it
            result['data'] = gzip.GzipFile(fileobj=StringIO(result['data']])).read()
    if hasattr(f, 'url'):
        result['url'] = f.url
        result['status'] = 200
    if hasattr(f, 'status'):
        result['status'] = f.status
    f.close()
    return result
```

Further reading

- Paul Prescod believes that pure HTTP web services are the future of the Internet (http://webservices.xml.com/pub/a/ws/2002/02/06/rest.html).

11.2. How not to fetch data over HTTP

Let's say you want to download a resource over HTTP, such as a syndicated Atom feed. But you don't just want to download it once; you want to download it over and over again, every hour, to get the latest news from the site that's offering the news feed. Let's do it the quick−and−dirty way first, and then see how you can do better.

Example 11.2. Downloading a feed the quick−and−dirty way

```
>>> import urllib
>>> data = urllib.urlopen('http://diveintomark.org/xml/atom.xml').read()    ❶
>>> print data
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/"/>
  <-- rest of feed omitted for brevity -->
```

❶ Downloading anything over HTTP is incredibly easy in Python; in fact, it's a one−liner. The `urllib` module has a handy `urlopen` function that takes the address of the page you want, and returns a file−like object that you can just `read()` from to get the full contents of the page. It just can't get much easier.

So what's wrong with this? Well, for a quick one−off during testing or development, there's nothing wrong with it. I do it all the time. I wanted the contents of the feed, and I got the contents of the feed. The same technique works for any web page. But once you start thinking in terms of a web service that you want to access on a regular basis −− and remember, you said you were planning on retrieving this syndicated feed once an hour −− then you're being inefficient, and you're being rude.

Let's talk about some of the basic features of HTTP.

11.3. Features of HTTP

There are five important features of HTTP which you should support.

11.3.1. `User−Agent`

The `User−Agent` is simply a way for a client to tell a server who it is when it requests a web page, a syndicated feed, or any sort of web service over HTTP. When the client requests a resource, it should always announce who it is, as specifically as possible. This allows the server−side administrator to get in touch with the client−side developer if anything is going fantastically wrong.

By default, Python sends a generic `User−Agent`: `Python−urllib/1.15`. In the next section, you'll see how to change this to something more specific.

11.3.2. Redirects

Sometimes resources move around. Web sites get reorganized, pages move to new addresses. Even web services can reorganize. A syndicated feed at `http://example.com/index.xml` might be moved to `http://example.com/xml/atom.xml`. Or an entire domain might move, as an organization expands and reorganizes; for instance, `http://www.example.com/index.xml` might be redirected to `http://server−farm−1.example.com/index.xml`.

Every time you request any kind of resource from an HTTP server, the server includes a status code in its response. Status code `200` means "everything's normal, here's the page you asked for". Status code `404` means "page not found". (You've probably seen 404 errors while browsing the web.)

HTTP has two different ways of signifying that a resource has moved. Status code `302` is a *temporary redirect*; it means "oops, that got moved over here temporarily" (and then gives the temporary address in a `Location:` header). Status code `301` is a *permanent redirect*; it means "oops, that got moved permanently" (and then gives the new address in a `Location:` header). If you get a `302` status code and a new address, the HTTP specification says you should use the new address to get what you asked for, but the next time you want to access the same resource, you should retry the old address. But if you get a `301` status code and a new address, you're supposed to use the new address from then on.

`urllib.urlopen` will automatically "follow" redirects when it receives the appropriate status code from the HTTP server, but unfortunately, it doesn't tell you when it does so. You'll end up getting data you asked for, but you'll never know that the underlying library "helpfully" followed a redirect for you. So you'll continue pounding away at the old address, and each time you'll get redirected to the new address. That's two round trips instead of one: not very efficient! Later in this chapter, you'll see how to work around this so you can deal with permanent redirects properly and efficiently.

11.3.3. `Last−Modified`/`If−Modified−Since`

Some data changes all the time. The home page of CNN.com is constantly updating every few minutes. On the other hand, the home page of Google.com only changes once every few weeks (when they put up a special holiday logo, or advertise a new service). Web services are no different; usually the server knows when the data you requested last changed, and HTTP provides a way for the server to include this last−modified date along with the data you requested.

If you ask for the same data a second time (or third, or fourth), you can tell the server the last−modified date that you got last time: you send an `If−Modified−Since` header with your request, with the date you got back from the server last time. If the data hasn't changed since then, the server sends back a special HTTP status code `304`, which

means "this data hasn't changed since the last time you asked for it". Why is this an improvement? Because when the server sends a `304`, *it doesn't re−send the data*. All you get is the status code. So you don't need to download the same data over and over again if it hasn't changed; the server assumes you have the data cached locally.

All modern web browsers support last−modified date checking. If you've ever visited a page, re−visited the same page a day later and found that it hadn't changed, and wondered why it loaded so quickly the second time −− this could be why. Your web browser cached the contents of the page locally the first time, and when you visited the second time, your browser automatically sent the last−modified date it got from the server the first time. The server simply says `304: Not Modified`, so your browser knows to load the page from its cache. Web services can be this smart too.

Python's URL library has no built−in support for last−modified date checking, but since you can add arbitrary headers to each request and read arbitrary headers in each response, you can add support for it yourself.

11.3.4. `ETag/If−None−Match`

ETags are an alternate way to accomplish the same thing as the last−modified date checking: don't re−download data that hasn't changed. The way it works is, the server sends some sort of hash of the data (in an `ETag` header) along with the data you requested. Exactly how this hash is determined is entirely up to the server. The second time you request the same data, you include the ETag hash in an `If−None−Match:` header, and if the data hasn't changed, the server will send you back a `304` status code. As with the last−modified date checking, the server *just* sends the `304`; it doesn't send you the same data a second time. By including the ETag hash in your second request, you're telling the server that there's no need to re−send the same data if it still matches this hash, since you still have the data from the last time.

Python's URL library has no built−in support for ETags, but you'll see how to add it later in this chapter.

11.3.5. Compression

The last important HTTP feature is gzip compression. When you talk about HTTP web services, you're almost always talking about moving XML back and forth over the wire. XML is text, and quite verbose text at that, and text generally compresses well. When you request a resource over HTTP, you can ask the server that, if it has any new data to send you, to please send it in compressed format. You include the `Accept−encoding: gzip` header in your request, and if the server supports compression, it will send you back gzip−compressed data and mark it with a `Content−encoding: gzip` header.

Python's URL library has no built−in support for gzip compression per se, but you can add arbitrary headers to the request. And Python comes with a separate `gzip` module, which has functions you can use to decompress the data yourself.

Note that our little one−line script to download a syndicated feed did not support any of these HTTP features. Let's see how you can improve it.

11.4. Debugging HTTP web services

First, let's turn on the debugging features of Python's HTTP library and see what's being sent over the wire. This will be useful throughout the chapter, as you add more and more features.

Example 11.3. Debugging HTTP

```
>>> import httplib
>>> httplib.HTTPConnection.debuglevel = 1             ➊
```

```
>>> import urllib
>>> feeddata = urllib.urlopen('http://diveintomark.org/xml/atom.xml').read()
connect: (diveintomark.org, 80)                                ❷
send: '
GET /xml/atom.xml HTTP/1.0                                     ❸
Host: diveintomark.org                                         ❹
User-agent: Python-urllib/1.15                                 ❺
'
reply: 'HTTP/1.1 200 OK\r\n'                                   ❻
header: Date: Wed, 14 Apr 2004 22:27:30 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT           ❼
header: ETag: "e8284-68e0-4de30f80"                            ❽
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close
```

❶ `urllib` relies on another standard Python library, `httplib`. Normally you don't need to `import httplib` directly (`urllib` does that automatically), but you will here so you can set the debugging flag on the `HTTPConnection` class that `urllib` uses internally to connect to the HTTP server. This is an incredibly useful technique. Some other Python libraries have similar debug flags, but there's no particular standard for naming them or turning them on; you need to read the documentation of each library to see if such a feature is available.

❷ Now that the debugging flag is set, information on the the HTTP request and response is printed out in real time. The first thing it tells you is that you're connecting to the server `diveintomark.org` on port 80, which is the standard port for HTTP.

❸ When you request the Atom feed, `urllib` sends three lines to the server. The first line specifies the HTTP verb you're using, and the path of the resource (minus the domain name). All the requests in this chapter will use `GET`, but in the next chapter on SOAP, you'll see that it uses `POST` for everything. The basic syntax is the same, regardless of the verb.

❹ The second line is the `Host` header, which specifies the domain name of the service you're accessing. This is important, because a single HTTP server can host multiple separate domains. My server currently hosts 12 domains; other servers can host hundreds or even thousands.

❺ The third line is the `User-Agent` header. What you see here is the generic `User-Agent` that the `urllib` library adds by default. In the next section, you'll see how to customize this to be more specific.

❻ The server replies with a status code and a bunch of headers (and possibly some data, which got stored in the `feeddata` variable). The status code here is `200`, meaning "everything's normal, here's the data you requested". The server also tells you the date it responded to your request, some information about the server itself, and the content type of the data it's giving you. Depending on your application, this might be useful, or not. It's certainly reassuring that you thought you were asking for an Atom feed, and lo and behold, you're getting an Atom feed (`application/atom+xml`, which is the registered content type for Atom feeds).

❼ The server tells you when this Atom feed was last modified (in this case, about 13 minutes ago). You can send this date back to the server the next time you request the same feed, and the server can do last−modified checking.

❽ The server also tells you that this Atom feed has an ETag hash of `"e8284-68e0-4de30f80"`. The hash doesn't mean anything by itself; there's nothing you can do with it, except send it back to the server the next time you request this same feed. Then the server can use it to tell you if the data has changed or not.

11.5. Setting the `User−Agent`

The first step to improving your HTTP web services client is to identify yourself properly with a `User−Agent`. To do that, you need to move beyond the basic `urllib` and dive into `urllib2`.

Example 11.4. Introducing `urllib2`

```
>>> import httplib
>>> httplib.HTTPConnection.debuglevel = 1                             (1)
>>> import urllib2
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml') (2)
>>> opener = urllib2.build_opener()                                   (3)
>>> feeddata = opener.open(request).read()                            (4)
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User−agent: Python−urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Wed, 14 Apr 2004 23:23:12 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content−Type: application/atom+xml
header: Last−Modified: Wed, 14 Apr 2004 22:14:38 GMT
header: ETag: "e8284−68e0−4de30f80"
header: Accept−Ranges: bytes
header: Content−Length: 26848
header: Connection: close
```

(1) If you still have your Python IDE open from the previous section's example, you can skip this, but this turns on HTTP debugging so you can see what you're actually sending over the wire, and what gets sent back.

(2) Fetching an HTTP resource with `urllib2` is a three−step process, for good reasons that will become clear shortly. The first step is to create a `Request` object, which takes the URL of the resource you'll eventually get around to retrieving. Note that this step doesn't actually retrieve anything yet.

(3) The second step is to build a URL opener. This can take any number of handlers, which control how responses are handled. But you can also build an opener without any custom handlers, which is what you're doing here. You'll see how to define and use custom handlers later in this chapter when you explore redirects.

(4) The final step is to tell the opener to open the URL, using the `Request` object you created. As you can see from all the debugging information that gets printed, this step actually retrieves the resource and stores the returned data in `feeddata`.

Example 11.5. Adding headers with the `Request`

```
>>> request                                                   (1)
<urllib2.Request instance at 0x00250AA8>
>>> request.get_full_url()
http://diveintomark.org/xml/atom.xml
>>> request.add_header('User−Agent',
...     'OpenAnything/1.0 +http://diveintopython.org/')       (2)
>>> feeddata = opener.open(request).read()                    (3)
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User−agent: OpenAnything/1.0 +http://diveintopython.org/      (4)
'
```

```
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Wed, 14 Apr 2004 23:45:17 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT
header: ETag: "e8284-68e0-4de30f80"
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close
```

1. You're continuing from the previous example; you've already created a `Request` object with the URL you want to access.
2. Using the `add_header` method on the `Request` object, you can add arbitrary HTTP headers to the request. The first argument is the header, the second is the value you're providing for that header. Convention dictates that a `User-Agent` should be in this specific format: an application name, followed by a slash, followed by a version number. The rest is free-form, and you'll see a lot of variations in the wild, but somewhere it should include a URL of your application. The `User-Agent` is usually logged by the server along with other details of your request, and including a URL of your application allows server administrators looking through their access logs to contact you if something is wrong.
3. The `opener` object you created before can be reused too, and it will retrieve the same feed again, but with your custom `User-Agent` header.
4. And here's you sending your custom `User-Agent`, in place of the generic one that Python sends by default. If you look closely, you'll notice that you defined a `User-Agent` header, but you actually sent a `User-agent` header. See the difference? `urllib2` changed the case so that only the first letter was capitalized. It doesn't really matter; HTTP specifies that header field names are completely case-insensitive.

11.6. Handling `Last-Modified` and `ETag`

Now that you know how to add custom HTTP headers to your web service requests, let's look at adding support for `Last-Modified` and `ETag` headers.

These examples show the output with debugging turned off. If you still have it turned on from the previous section, you can turn it off by setting `httplib.HTTPConnection.debuglevel = 0`. Or you can just leave debugging on, if that helps you.

Example 11.6. Testing `Last-Modified`

```
>>> import urllib2
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> opener = urllib2.build_opener()
>>> firstdatastream = opener.open(request)
>>> firstdatastream.headers.dict                          (1)
{'date': 'Thu, 15 Apr 2004 20:42:41 GMT',
 'server': 'Apache/2.0.49 (Debian GNU/Linux)',
 'content-type': 'application/atom+xml',
 'last-modified': 'Thu, 15 Apr 2004 19:45:21 GMT',
 'etag': '"e842a-3e53-55d97640"',
 'content-length': '15955',
 'accept-ranges': 'bytes',
 'connection': 'close'}
>>> request.add_header('If-Modified-Since',
...     firstdatastream.headers.get('Last-Modified'))     (2)
>>> seconddatastream = opener.open(request)               (3)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\lib\urllib2.py", line 326, in open
    '_open', req)
  File "c:\python23\lib\urllib2.py", line 306, in _call_chain
    result = func(*args)
  File "c:\python23\lib\urllib2.py", line 901, in http_open
    return self.do_open(httplib.HTTP, req)
  File "c:\python23\lib\urllib2.py", line 895, in do_open
    return self.parent.error('http', req, fp, code, msg, hdrs)
  File "c:\python23\lib\urllib2.py", line 352, in error
    return self._call_chain(*args)
  File "c:\python23\lib\urllib2.py", line 306, in _call_chain
    result = func(*args)
  File "c:\python23\lib\urllib2.py", line 412, in http_error_default
    raise HTTPError(req.get_full_url(), code, msg, hdrs, fp)
urllib2.HTTPError: HTTP Error 304: Not Modified
```

❶ Remember all those HTTP headers you saw printed out when you turned on debugging? This is how you can get access to them programmatically: `firstdatastream.headers` is an object that acts like a dictionary and allows you to get any of the individual headers returned from the HTTP server.

❷ On the second request, you add the `If-Modified-Since` header with the last−modified date from the first request. If the data hasn't changed, the server should return a `304` status code.

❸ Sure enough, the data hasn't changed. You can see from the traceback that `urllib2` throws a special exception, `HTTPError`, in response to the `304` status code. This is a little unusual, and not entirely helpful. After all, it's not an error; you specifically asked the server not to send you any data if it hadn't changed, and the data didn't change, so the server told you it wasn't sending you any data. That's not an error; that's exactly what you were hoping for.

`urllib2` also raises an `HTTPError` exception for conditions that you would think of as errors, such as `404` (page not found). In fact, it will raise `HTTPError` for *any* status code other than `200` (OK), `301` (permanent redirect), or `302` (temporary redirect). It would be more helpful for your purposes to capture the status code and simply return it, without throwing an exception. To do that, you'll need to define a custom URL handler.

Example 11.7. Defining URL handlers

This custom URL handler is part of `openanything.py`.

```
class DefaultErrorHandler(urllib2.HTTPDefaultErrorHandler):    ❶
    def http_error_default(self, req, fp, code, msg, headers): ❷
        result = urllib2.HTTPError(
            req.get_full_url(), code, msg, headers, fp)
        result.status = code                                   ❸
        return result
```

❶ `urllib2` is designed around URL handlers. Each handler is just a class that can define any number of methods. When something happens −− like an HTTP error, or even a `304` code −− `urllib2` introspects into the list of defined handlers for a method that can handle it. You used a similar introspection in Chapter 9, *XML Processing* to define handlers for different node types, but `urllib2` is more flexible, and introspects over as many handlers as are defined for the current request.

❷ `urllib2` searches through the defined handlers and calls the `http_error_default` method when it encounters a `304` status code from the server. By defining a custom error handler, you can prevent `urllib2` from raising an exception. Instead, you create the `HTTPError` object, but return it instead of raising it.

❸ This is the key part: before returning, you save the status code returned by the HTTP server. This will allow you easy access to it from the calling program.

Example 11.8. Using custom URL handlers

```
>>> request.headers                                    ❶
{'If-modified-since': 'Thu, 15 Apr 2004 19:45:21 GMT'}
>>> import openanything
>>> opener = urllib2.build_opener(
...     openanything.DefaultErrorHandler())            ❷
>>> seconddatastream = opener.open(request)
>>> seconddatastream.status                            ❸
304
>>> seconddatastream.read()                            ❹
''
```

❶ You're continuing the previous example, so the `Request` object is already set up, and you've already added the `If-Modified-Since` header.

❷ This is the key: now that you've defined your custom URL handler, you need to tell `urllib2` to use it. Remember how I said that `urllib2` broke up the process of accessing an HTTP resource into three steps, and for good reason? This is why building the URL opener is its own step, because you can build it with your own custom URL handlers that override `urllib2`'s default behavior.

❸ Now you can quietly open the resource, and what you get back is an object that, along with the usual headers (use `seconddatastream.headers.dict` to acess them), also contains the HTTP status code. In this case, as you expected, the status is `304`, meaning this data hasn't changed since the last time you asked for it.

❹ Note that when the server sends back a `304` status code, it doesn't re−send the data. That's the whole point: to save bandwidth by not re−downloading data that hasn't changed. So if you actually want that data, you'll need to cache it locally the first time you get it.

Handling `ETag` works much the same way, but instead of checking for `Last-Modified` and sending `If-Modified-Since`, you check for `ETag` and send `If-None-Match`. Let's start with a fresh IDE session.

Example 11.9. Supporting `ETag`/`If-None-Match`

```
>>> import urllib2, openanything
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> opener = urllib2.build_opener(
...     openanything.DefaultErrorHandler())
>>> firstdatastream = opener.open(request)
>>> firstdatastream.headers.get('ETag')                 ❶
'"e842a-3e53-55d97640"'
>>> firstdata = firstdatastream.read()
>>> print firstdata                                     ❷
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/"/>
  <-- rest of feed omitted for brevity -->
>>> request.add_header('If-None-Match',
...     firstdatastream.headers.get('ETag'))            ❸
>>> seconddatastream = opener.open(request)
>>> seconddatastream.status                             ❹
304
>>> seconddatastream.read()                             ❺
''
```

❶

Using the `firstdatastream.headers` pseudo−dictionary, you can get the `ETag` returned from the server. (What happens if the server didn't send back an `ETag`? Then this line would return `None`.)

❷ OK, you got the data.

❸ Now set up the second call by setting the `If−None−Match` header to the `ETag` you got from the first call.

❹ The second call succeeds quietly (without throwing an exception), and once again you see that the server has sent back a `304` status code. Based on the `ETag` you sent the second time, it knows that the data hasn't changed.

❺ Regardless of whether the `304` is triggered by `Last−Modified` date checking or `ETag` hash matching, you'll never get the data along with the `304`. That's the whole point.

In these examples, the HTTP server has supported both `Last−Modified` and `ETag` headers, but not all servers do. As a web services client, you should be prepared to support both, but you must code defensively in case a server only supports one or the other, or neither.

11.7. Handling redirects

You can support permanent and temporary redirects using a different kind of custom URL handler.

First, let's see why a redirect handler is necessary in the first place.

Example 11.10. Accessing web services without a redirect handler

```
>>> import urllib2, httplib
>>> httplib.HTTPConnection.debuglevel = 1            ❶
>>> request = urllib2.Request(
...     'http://diveintomark.org/redir/example301.xml') ❷
>>> opener = urllib2.build_opener()
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /redir/example301.xml HTTP/1.0
Host: diveintomark.org
User−agent: Python−urllib/2.1
'
reply: 'HTTP/1.1 301 Moved Permanently\r\n'             ❸
header: Date: Thu, 15 Apr 2004 22:06:25 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml  ❹
header: Content−Length: 338
header: Connection: close
header: Content−Type: text/html; charset=iso−8859−1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0                              ❺
Host: diveintomark.org
User−agent: Python−urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:06:25 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last−Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a−3e53−55d97640"
header: Accept−Ranges: bytes
header: Content−Length: 15955
header: Connection: close
```

```
header: Content-Type: application/atom+xml
>>> f.url                                                     ❻
'http://diveintomark.org/xml/atom.xml'
>>> f.headers.dict
{'content-length': '15955',
'accept-ranges': 'bytes',
'server': 'Apache/2.0.49 (Debian GNU/Linux)',
'last-modified': 'Thu, 15 Apr 2004 19:45:21 GMT',
'connection': 'close',
'etag': '"e842a-3e53-55d97640"',
'date': 'Thu, 15 Apr 2004 22:06:25 GMT',
'content-type': 'application/atom+xml'}
>>> f.status
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: addinfourl instance has no attribute 'status'
```

❶ You'll be better able to see what's happening if you turn on debugging.

❷ This is a URL which I have set up to permanently redirect to my Atom feed at `http://diveintomark.org/xml/atom.xml`.

❸ Sure enough, when you try to download the data at that address, the server sends back a `301` status code, telling you that the resource has moved permanently.

❹ The server also sends back a `Location:` header that gives the new address of this data.

❺ `urllib2` notices the redirect status code and automatically tries to retrieve the data at the new location specified in the `Location:` header.

❻ The object you get back from the `opener` contains the new permanent address and all the headers returned from the second request (retrieved from the new permanent address). But the status code is missing, so you have no way of knowing programmatically whether this redirect was temporary or permanent. And that matters very much: if it was a temporary redirect, then you should continue to ask for the data at the old location. But if it was a permanent redirect (as this was), you should ask for the data at the new location from now on.

This is suboptimal, but easy to fix. `urllib2` doesn't behave exactly as you want it to when it encounters a `301` or `302`, so let's override its behavior. How? With a custom URL handler, just like you did to handle `304` codes.

Example 11.11. Defining the redirect handler

This class is defined in `openanything.py`.

```
class SmartRedirectHandler(urllib2.HTTPRedirectHandler):     ❶
    def http_error_301(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_301( ❷
            self, req, fp, code, msg, headers)
        result.status = code                                 ❸
        return result

    def http_error_302(self, req, fp, code, msg, headers):   ❹
        result = urllib2.HTTPRedirectHandler.http_error_302(
            self, req, fp, code, msg, headers)
        result.status = code
        return result
```

❶ Redirect behavior is defined in `urllib2` in a class called `HTTPRedirectHandler`. You don't want to completely override the behavior, you just want to extend it a little, so you'll subclass `HTTPRedirectHandler` so you can call the ancestor class to do all the hard work.

❷

When it encounters a `301` status code from the server, `urllib2` will search through its handlers and call the `http_error_301` method. The first thing ours does is just call the `http_error_301` method in the ancestor, which handles the grunt work of looking for the `Location:` header and following the redirect to the new address.

❸ Here's the key: before you return, you store the status code (`301`), so that the calling program can access it later.

❹ Temporary redirects (status code `302`) work the same way: override the `http_error_302` method, call the ancestor, and save the status code before returning.

So what has this bought us? You can now build a URL opener with the custom redirect handler, and it will still automatically follow redirects, but now it will also expose the redirect status code.

Example 11.12. Using the redirect handler to detect permanent redirects

```
>>> request = urllib2.Request('http://diveintomark.org/redir/example301.xml')
>>> import openanything, httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> opener = urllib2.build_opener(
...     openanything.SmartRedirectHandler())                   ❶
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: 'GET /redir/example301.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 301 Moved Permanently\r\n'                    ❷
header: Date: Thu, 15 Apr 2004 22:13:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 338
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:13:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml

>>> f.status                                                   ❸
301
>>> f.url
'http://diveintomark.org/xml/atom.xml'
```

❶ First, build a URL opener with the redirect handler you just defined.

❷ You sent off a request, and you got a `301` status code in response. At this point, the `http_error_301` method gets called. You call the ancestor method, which follows the redirect and sends a request at the new location (`http://diveintomark.org/xml/atom.xml`).

3. This is the payoff: now, not only do you have access to the new URL, but you have access to the redirect status code, so you can tell that this was a permanent redirect. The next time you request this data, you should request it from the new location (`http://diveintomark.org/xml/atom.xml`, as specified in `f.url`). If you had stored the location in a configuration file or a database, you need to update that so you don't keep pounding the server with requests at the old address. It's time to update your address book.

The same redirect handler can also tell you that you *shouldn't* update your address book.

Example 11.13. Using the redirect handler to detect temporary redirects

```
>>> request = urllib2.Request(
...     'http://diveintomark.org/redir/example302.xml')          (1)
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /redir/example302.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 302 Found\r\n'                                  (2)
header: Date: Thu, 15 Apr 2004 22:18:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 314
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0                                       (3)
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:18:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml
>>> f.status                                                     (4)
302
>>> f.url
http://diveintomark.org/xml/atom.xml
```

1. This is a sample URL I've set up that is configured to tell clients to *temporarily* redirect to `http://diveintomark.org/xml/atom.xml`.
2. The server sends back a `302` status code, indicating a temporary redirect. The temporary new location of the data is given in the `Location:` header.
3. `urllib2` calls your `http_error_302` method, which calls the ancestor method of the same name in `urllib2.HTTPRedirectHandler`, which follows the redirect to the new location. Then your `http_error_302` method stores the status code (`302`) so the calling application can get it later.
4. And here you are, having successfully followed the redirect to `http://diveintomark.org/xml/atom.xml`. `f.status` tells you that this was a temporary redirect, which means that you should continue to request data from the original address (`http://diveintomark.org/redir/example302.xml`). Maybe it will redirect next time too, but

maybe not. Maybe it will redirect to a different address. It's not for you to say. The server said this redirect was only temporary, so you should respect that. And now you're exposing enough information that the calling application can respect that.

11.8. Handling compressed data

The last important HTTP feature you want to support is compression. Many web services have the ability to send data compressed, which can cut down the amount of data sent over the wire by 60% or more. This is especially true of XML web services, since XML data compresses very well.

Servers won't give you compressed data unless you tell them you can handle it.

Example 11.14. Telling the server you would like compressed data

```
>>> import urllib2, httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> request.add_header('Accept-encoding', 'gzip')        ❶
>>> opener = urllib2.build_opener()
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
Accept-encoding: gzip                                    ❷
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:24:39 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Vary: Accept-Encoding
header: Content-Encoding: gzip                           ❸
header: Content-Length: 6289                             ❹
header: Connection: close
header: Content-Type: application/atom+xml
```

❶ This is the key: once you've created your `Request` object, add an `Accept-encoding` header to tell the server you can accept gzip−encoded data. `gzip` is the name of the compression algorithm you're using. In theory there could be other compression algorithms, but `gzip` is the compression algorithm used by 99% of web servers.

❷ There's your header going across the wire.

❸ And here's what the server sends back: the `Content-Encoding: gzip` header means that the data you're about to receive has been gzip−compressed.

❹ The `Content-Length` header is the length of the compressed data, not the uncompressed data. As you'll see in a minute, the actual length of the uncompressed data was 15955, so gzip compression cut your bandwidth by over 60%!

Example 11.15. Decompressing the data

```
>>> compresseddata = f.read()                              ❶
>>> len(compresseddata)
6289
>>> import StringIO
```

```
>>> compressedstream = StringIO.StringIO(compresseddata)   ❷
>>> import gzip
>>> gzipper = gzip.GzipFile(fileobj=compressedstream)      ❸
>>> data = gzipper.read()                                  ❹
>>> print data                                             ❺
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/"/>
  <-- rest of feed omitted for brevity -->
>>> len(data)
15955
```

❶ Continuing from the previous example, `f` is the file−like object returned from the URL opener. Using its `read()` method would ordinarily get you the uncompressed data, but since this data has been gzip−compressed, this is just the first step towards getting the data you really want.

❷ OK, this step is a little bit of messy workaround. Python has a `gzip` module, which reads (and actually writes) gzip−compressed files on disk. But you don't have a file on disk, you have a gzip−compressed buffer in memory, and you don't want to write out a temporary file just so you can uncompress it. So what you're going to do is create a file−like object out of the in−memory data (`compresseddata`), using the `StringIO` module. You first saw the `StringIO` module in the previous chapter, but now you've found another use for it.

❸ Now you can create an instance of `GzipFile`, and tell it that its "file" is the file−like object `compressedstream`.

❹ This is the line that does all the actual work: "reading" from `GzipFile` will decompress the data. Strange? Yes, but it makes sense in a twisted kind of way. `gzipper` is a file−like object which represents a gzip−compressed file. That "file" is not a real file on disk, though; `gzipper` is really just "reading" from the file−like object you created with `StringIO` to wrap the compressed data, which is only in memory in the variable `compresseddata`. And where did that compressed data come from? You originally downloaded it from a remote HTTP server by "reading" from the file−like object you built with `urllib2.build_opener`. And amazingly, this all just works. Every step in the chain has no idea that the previous step is faking it.

❺ Look ma, real data. (15955 bytes of it, in fact.)

"But wait!" I hear you cry. "This could be even easier!" I know what you're thinking. You're thinking that `opener.open` returns a file−like object, so why not cut out the `StringIO` middleman and just pass `f` directly to `GzipFile`? OK, maybe you weren't thinking that, but don't worry about it, because it doesn't work.

Example 11.16. Decompressing the data directly from the server

```
>>> f = opener.open(request)                          ❶
>>> f.headers.get('Content-Encoding')                 ❷
'gzip'
>>> data = gzip.GzipFile(fileobj=f).read()            ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\lib\gzip.py", line 217, in read
    self._read(readsize)
  File "c:\python23\lib\gzip.py", line 252, in _read
    pos = self.fileobj.tell()   # Save current position
AttributeError: addinfourl instance has no attribute 'tell'
```

❶ Continuing from the previous example, you already have a `Request` object set up with an `Accept-encoding: gzip` header.

❷ Simply opening the request will get you the headers (though not download any data yet). As you can see from the returned `Content-Encoding` header, this data has been sent gzip−compressed.

❸ Since `opener.open` returns a file−like object, and you know from the headers that when you read it, you're going to get gzip−compressed data, why not simply pass that file−like object directly to `GzipFile`? As you "read" from the `GzipFile` instance, it will "read" compressed data from the remote HTTP server and decompress it on the fly. It's a good idea, but unfortunately it doesn't work. Because of the way gzip compression works, `GzipFile` needs to save its position and move forwards and backwards through the compressed file. This doesn't work when the "file" is a stream of bytes coming from a remote server; all you can do with it is retrieve bytes one at a time, not move back and forth through the data stream. So the inelegant hack of using `StringIO` is the best solution: download the compressed data, create a file−like object out of it with `StringIO`, and then decompress the data from that.

11.9. Putting it all together

You've seen all the pieces for building an intelligent HTTP web services client. Now let's see how they all fit together.

Example 11.17. The `openanything` function

This function is defined in `openanything.py`.

```
def openAnything(source, etag=None, lastmodified=None, agent=USER_AGENT):
    # non−HTTP code omitted for brevity
    if urlparse.urlparse(source)[0] == 'http':                                    ❶
        # open URL with urllib2
        request = urllib2.Request(source)
        request.add_header('User−Agent', agent)                                   ❷
        if etag:
            request.add_header('If−None−Match', etag)                             ❸
        if lastmodified:
            request.add_header('If−Modified−Since', lastmodified)                 ❹
        request.add_header('Accept−encoding', 'gzip')                             ❺
        opener = urllib2.build_opener(SmartRedirectHandler(), DefaultErrorHandler()) ❻
        return opener.open(request)                                               ❼
```

❶ `urlparse` is a handy utility module for, you guessed it, parsing URLs. It's primary function, also called `urlparse`, takes a URL and splits it into a tuple of (scheme, domain, path, params, query string parameters, and fragment identifier). Of these, the only thing you care about is the scheme, to make sure that you're dealing with an HTTP URL (which `urllib2` can handle).

❷ You identify yourself to the HTTP server with the `User−Agent` passed in by the calling function. If no `User−Agent` was specified, you use a default one defined earlier in the `openanything.py` module. You never use the default one defined by `urllib2`.

❸ If an `ETag` hash was given, send it in the `If−None−Match` header.

❹ If a last−modified date was given, send it in the `If−Modified−Since` header.

❺ Tell the server you would like compressed data if possible.

❻ Build a URL opener that uses *both* of the custom URL handlers: `SmartRedirectHandler` for handling `301` and `302` redirects, and `DefaultErrorHandler` for handling `304`, `404`, and other error conditions gracefully.

❼ That's it! Open the URL and return a file−like object to the caller.

Example 11.18. The `fetch` function

This function is defined in `openanything.py`.

```
def fetch(source, etag=None, last_modified=None, agent=USER_AGENT):
    '''Fetch data and metadata from a URL, file, stream, or string'''
    result = {}
    f = openAnything(source, etag, last_modified, agent)                ❶
    result['data'] = f.read()                                           ❷
    if hasattr(f, 'headers'):
        # save ETag, if the server sent one
        result['etag'] = f.headers.get('ETag')                          ❸
        # save Last-Modified header, if the server sent one
        result['lastmodified'] = f.headers.get('Last-Modified')         ❹
        if f.headers.get('content-encoding', '') == 'gzip':             ❺
            # data came back gzip-compressed, decompress it
            result['data'] = gzip.GzipFile(fileobj=StringIO(result['data']])).read()
    if hasattr(f, 'url'):                                               ❻
        result['url'] = f.url
        result['status'] = 200
    if hasattr(f, 'status'):                                            ❼
        result['status'] = f.status
    f.close()
    return result
```

❶ First, you call the `openAnything` function with a URL, `ETag` hash, `Last-Modified` date, and `User-Agent`.

❷ Read the actual data returned from the server. This may be compressed; if so, you'll decompress it later.

❸ Save the `ETag` hash returned from the server, so the calling application can pass it back to you next time, and you can pass it on to `openAnything`, which can stick it in the `If-None-Match` header and send it to the remote server.

❹ Save the `Last-Modified` date too.

❺ If the server says that it sent compressed data, decompress it.

❻ If you got a URL back from the server, save it, and assume that the status code is `200` until you find out otherwise.

❼ If one of the custom URL handlers captured a status code, then save that too.

Example 11.19. Using `openanything.py`

```
>>> import openanything
>>> useragent = 'MyHTTPWebServicesApp/1.0'
>>> url = 'http://diveintopython.org/redir/example301.xml'
>>> params = openanything.fetch(url, agent=useragent)              ❶
>>> params                                                         ❷
{'url': 'http://diveintomark.org/xml/atom.xml',
'lastmodified': 'Thu, 15 Apr 2004 19:45:21 GMT',
'etag': '"e842a-3e53-55d97640"',
'status': 301,
'data': '<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
<-- rest of data omitted for brevity -->'}
>>> if params['status'] == 301:                                    ❸
...     url = params['url']
>>> newparams = openanything.fetch(
...     url, params['etag'], params['lastmodified'], useragent)    ❹
>>> newparams
{'url': 'http://diveintomark.org/xml/atom.xml',
'lastmodified': None,
'etag': '"e842a-3e53-55d97640"',
'status': 304,
```

```
'data': ''}                                                   ⑤
```

1. The very first time you fetch a resource, you don't have an `ETag` hash or `Last-Modified` date, so you'll leave those out. (They're optional parameters.)
2. What you get back is a dictionary of several useful headers, the HTTP status code, and the actual data returned from the server. `openanything` handles the gzip compression internally; you don't care about that at this level.
3. If you ever get a `301` status code, that's a permanent redirect, and you need to update your URL to the new address.
4. The second time you fetch the same resource, you have all sorts of information to pass back: a (possibly updated) URL, the `ETag` from the last time, the `Last-Modified` date from the last time, and of course your `User-Agent`.
5. What you get back is again a dictionary, but the data hasn't changed, so all you got was a `304` status code and no data.

11.10. Summary

The `openanything.py` and its functions should now make perfect sense.

There are 5 important features of HTTP web services that every client should support:

- Identifying your application by setting a proper `User-Agent`.
- Handling permanent redirects properly.
- Supporting `Last-Modified` date checking to avoid re−downloading data that hasn't changed.
- Supporting `ETag` hashes to avoid re−downloading data that hasn't changed.
- Supporting gzip compression to reduce bandwidth even when data *has* changed.

Chapter 12. SOAP Web Services

Chapter 11 focused on document−oriented web services over HTTP. The "input parameter" was the URL, and the "return value" was an actual XML document which it was your responsibility to parse.

This chapter will focus on SOAP web services, which take a more structured approach. Rather than dealing with HTTP requests and XML documents directly, SOAP allows you to simulate calling functions that return native data types. As you will see, the illusion is almost perfect; you can "call" a function through a SOAP library, with the standard Python calling syntax, and the function appears to return Python objects and values. But under the covers, the SOAP library has actually performed a complex transaction involving multiple XML documents and a remote server.

SOAP is a complex specification, and it is somewhat misleading to say that SOAP is all about calling remote functions. Some people would pipe up to add that SOAP allows for one−way asynchronous message passing, and document−oriented web services. And those people would be correct; SOAP can be used that way, and in many different ways. But this chapter will focus on so−called "RPC−style" SOAP −− calling a remote function and getting results back.

12.1. Diving In

You use Google, right? It's a popular search engine. Have you ever wished you could programmatically access Google search results? Now you can. Here is a program to search Google from Python.

Example 12.1. `search.py`

```
from SOAPpy import WSDL

# you'll need to configure these two values;
# see http://www.google.com/apis/
WSDLFILE = '/path/to/copy/of/GoogleSearch.wsdl'
APIKEY = 'YOUR_GOOGLE_API_KEY'

_server = WSDL.Proxy(WSDLFILE)
def search(q):
    """Search Google and return list of {title, link, description}"""
    results = _server.doGoogleSearch(
        APIKEY, q, 0, 10, False, "", False, "", "utf-8", "utf-8")
    return [{"title": r.title.encode("utf-8"),
             "link": r.URL.encode("utf-8"),
             "description": r.snippet.encode("utf-8")}
            for r in results.resultElements]

if __name__ == '__main__':
    import sys
    for r in search(sys.argv[1])[:5]:
        print r['title']
        print r['link']
        print r['description']
        print
```

You can import this as a module and use it from a larger program, or you can run the script from the command line. On the command line, you give the search query as a command−line argument, and it prints out the URL, title, and description of the top five Google search results.

Here is the sample output for a search for the word "python".

Example 12.2. Sample Usage of `search.py`

```
C:\diveintopython\common\py> python search.py "python"
<b>Python</b> Programming Language
http://www.python.org/
Home page for <b>Python</b>, an interpreted, interactive, object−oriented,
extensible<br> programming language. <b>...</b> <b>Python</b>
is OSI Certified Open Source: OSI Certified.

<b>Python</b> Documentation Index
http://www.python.org/doc/
 <b>...</b> New−style classes (aka descrintro). Regular expressions. Database
API. Email Us.<br> docs@<b>python</b>.org. (c) 2004. <b>Python</b>
Software Foundation. <b>Python</b> Documentation. <b>...</b>

Download <b>Python</b> Software
http://www.python.org/download/
Download Standard <b>Python</b> Software. <b>Python</b> 2.3.3 is the
current production<br> version of <b>Python</b>. <b>...</b>
<b>Python</b> is OSI Certified Open Source:

Pythonline
http://www.pythonline.com/


Dive Into <b>Python</b>
http://diveintopython.org/
Dive Into <b>Python</b>. <b>Python</b> from novice to pro. Find:
<b>...</b> It is also available in multiple<br> languages. Read
Dive Into <b>Python</b>. This book is still being written. <b>...</b>
```

Further Reading on SOAP

- http://www.xmethods.net/ is a repository of public access SOAP web services.
- The SOAP specification (http://www.w3.org/TR/soap/) is surprisingly readable, if you like that sort of thing.

12.2. Installing the SOAP Libraries

Unlike the other code in this book, this chapter relies on libraries that do not come pre−installed with Python.

Before you can dive into SOAP web services, you'll need to install three libraries: PyXML, fpconst, and SOAPpy.

12.2.1. Installing PyXML

The first library you need is PyXML, an advanced set of XML libraries that provide more functionality than the built−in XML libraries we studied in Chapter 9.

Procedure 12.1.

Here is the procedure for installing PyXML:

1. Go to http://pyxml.sourceforge.net/, click Downloads, and download the latest version for your operating system.
2. If you are using Windows, there are several choices. Make sure to download the version of PyXML that matches the version of Python you are using.
3. Double−click the installer. If you download PyXML 0.8.3 for Windows and Python 2.3, the installer program

will be `PyXML−0.8.3.win32−py2.3.exe`.

4. Step through the installer program.
5. After the installation is complete, close the installer. There will not be any visible indication of success (no programs installed on the Start Menu or shortcuts installed on the desktop). PyXML is simply a collection of XML libraries used by other programs.

To verify that you installed PyXML correctly, run your Python IDE and check the version of the XML libraries you have installed, as shown here.

Example 12.3. Verifying PyXML Installation

```
>>> import xml
>>> xml.__version__
'0.8.3'
```

This version number should match the version number of the PyXML installer program you downloaded and ran.

12.2.2. Installing fpconst

The second library you need is fpconst, a set of constants and functions for working with IEEE754 double−precision special values. This provides support for the special values Not−a−Number (NaN), Positive Infinity (Inf), and Negative Infinity (−Inf), which are part of the SOAP datatype specification.

Procedure 12.2.

Here is the procedure for installing fpconst:

1. Download the latest version of fpconst from http://www.analytics.washington.edu/statcomp/projects/rzope/fpconst/.
2. There are two downloads available, one in `.tar.gz` format, the other in `.zip` format. If you are using Windows, download the `.zip` file; otherwise, download the `.tar.gz` file.
3. Decompress the downloaded file. On Windows XP, you can right−click on the file and choose Extract All; on earlier versions of Windows, you will need a third−party program such as WinZip. On Mac OS X, you can double−click the compressed file to decompress it with Stuffit Expander.
4. Open a command prompt and navigate to the directory where you decompressed the fpconst files.
5. Type **`python setup.py install`** to run the installation program.

To verify that you installed fpconst correctly, run your Python IDE and check the version number.

Example 12.4. Verifying fpconst Installation

```
>>> import fpconst
>>> fpconst.__version__
'0.6.0'
```

This version number should match the version number of the fpconst archive you downloaded and installed.

12.2.3. Installing SOAPpy

The third and final requirement is the SOAP library itself: SOAPpy.

Procedure 12.3.

Here is the procedure for installing SOAPpy:

1. Go to http://pywebsvcs.sourceforge.net/ and select Latest Official Release under the SOAPpy section.
2. There are two downloads available. If you are using Windows, download the `.zip` file; otherwise, download the `.tar.gz` file.
3. Decompress the downloaded file, just as you did with fpconst.
4. Open a command prompt and navigate to the directory where you decompressed the SOAPpy files.
5. Type **`python setup.py install`** to run the installation program.

To verify that you installed SOAPpy correctly, run your Python IDE and check the version number.

Example 12.5. Verifying SOAPpy Installation

```
>>> import SOAPpy
>>> SOAPpy.__version__
'0.11.4'
```

This version number should match the version number of the SOAPpy archive you downloaded and installed.

12.3. First Steps with SOAP

The heart of SOAP is the ability to call remote functions. There are a number of public access SOAP servers that provide simple functions for demonstration purposes.

The most popular public access SOAP server is http://www.xmethods.net/. This example uses a demonstration function that takes a United States zip code and returns the current temperature in that region.

Example 12.6. Getting the Current Temperature

```
>>> from SOAPpy import SOAPProxy                   ❶
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> namespace = 'urn:xmethods-Temperature'  ❷
>>> server = SOAPProxy(url, namespace)      ❸
>>> server.getTemp('27502')                 ❹
80.0
```

❶ You access the remote SOAP server through a proxy class, `SOAPProxy`. The proxy handles all the internals of SOAP for you, including creating the XML request document out of the function name and argument list, sending the request over HTTP to the remote SOAP server, parsing the XML response document, and creating native Python values to return. You'll see what these XML documents look like in the next section.

❷ Every SOAP service has a URL which handles all the requests. The same URL is used for all function calls. This particular service only has a single function, but later in this chapter you'll see examples of the Google API, which has several functions. The service URL is shared by all functions.Each SOAP service also has a namespace, which is defined by the server and is completely arbitrary. It's simply part of the configuration required to call SOAP methods. It allows the server to share a single service URL and route requests between several unrelated services. It's like dividing Python modules into packages.

❸ You're creating the `SOAPProxy` with the service URL and the service namespace. This doesn't make any connection to the SOAP server; it simply creates a local Python object.

❹ Now with everything configured properly, you can actually call remote SOAP methods as if they were local functions. You pass arguments just like a normal function, and you get a return value just like a normal function. But under the covers, there's a heck of a lot going on.

Let's peek under those covers.

12.4. Debugging SOAP Web Services

The SOAP libraries provide an easy way to see what's going on behind the scenes.

Turning on debugging is a simple matter of setting two flags in the `SOAPProxy`'s configuration.

Example 12.7. Debugging SOAP Web Services

```
>>> from SOAPpy import SOAPProxy
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> n = 'urn:xmethods-Temperature'
>>> server = SOAPProxy(url, namespace=n)      ❶
>>> server.config.dumpSOAPOut = 1             ❷
>>> server.config.dumpSOAPIn = 1
>>> temperature = server.getTemp('27502')     ❸
*** Outgoing SOAP ******************************************************
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">27502</v1>
</ns1:getTemp>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************
*** Incoming SOAP ******************************************************
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">80.0</return>
</ns1:getTempResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************

>>> temperature
80.0
```

❶ First, create the `SOAPProxy` like normal, with the service URL and the namespace.

❷ Second, turn on debugging by setting `server.config.dumpSOAPIn` and `server.config.dumpSOAPOut`.

❸ Third, call the remote SOAP method as usual. The SOAP library will print out both the outgoing XML request

document, and the incoming XML response document. This is all the hard work that SOAPProxy is doing for you. Intimidating, isn't it? Let's break it down.

Most of the XML request document that gets sent to the server is just boilerplate. Ignore all the namespace declarations; they're going to be the same (or similar) for all SOAP calls. The heart of the "function call" is this fragment within the <Body> element:

```
<ns1:getTemp                                  ❶
  xmlns:ns1="urn:xmethods-Temperature"        ❷
  SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">27502</v1>          ❸
</ns1:getTemp>
```

❶ The element name is the function name, getTemp. SOAPProxy uses getattr as a dispatcher. Instead of calling separate local methods based on the method name, it actually uses the method name to construct the XML request document.

❷ The function's XML element is contained in a specific namespace, which is the namespace you specified when you created the SOAPProxy object. Don't worry about the SOAP-ENC:root; that's boilerplate too.

❸ The arguments of the function also got translated into XML. SOAPProxy introspects each argument to determine its datatype (in this case it's a string). The argument datatype goes into the xsi:type attribute, followed by the actual string value.

The XML return document is equally easy to understand, once you know what to ignore. Focus on this fragment within the <Body>:

```
<ns1:getTempResponse                              ❶
  xmlns:ns1="urn:xmethods-Temperature"            ❷
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">80.0</return>        ❸
</ns1:getTempResponse>
```

❶ The server wraps the function return value within a <getTempResponse> element. By convention, this wrapper element is the name of the function, plus Response. But it could really be almost anything; the important thing that SOAPProxy notices is not the element name, but the namespace.

❷ The server returns the response in the same namespace we used in the request, the same namespace we specified when we first create the SOAPProxy. Later in this chapter we'll see what happens if you forget to specify the namespace when creating the SOAPProxy.

❸ The return value is specified, along with its datatype (it's a float). SOAPProxy uses this explicit datatype to create a Python object of the correct native datatype and return it.

12.5. Introducing WSDL

The SOAPProxy class proxies local method calls and transparently turns then into invocations of remote SOAP methods. As you've seen, this is a lot of work, and SOAPProxy does it quickly and transparently. What it doesn't do is provide any means of method introspection.

Consider this: the previous two sections showed an example of calling a simple remote SOAP method with one argument and one return value, both of simple data types. This required knowing, and keeping track of, the service URL, the service namespace, the function name, the number of arguments, and the datatype of each argument. If any of these is missing or wrong, the whole thing falls apart.

That shouldn't come as a big surprise. If I wanted to call a local function, I would need to know what package or module it was in (the equivalent of service URL and namespace). I would need to know the correct function name and the correct number of arguments. Python deftly handles datatyping without explicit types, but I would still need to know how many argument to pass, and how many return values to expect.

The big difference is introspection. As you saw in Chapter 4, Python excels at letting you discover things about modules and functions at runtime. You can list the available functions within a module, and with a little work, drill down to individual function declarations and arguments.

WSDL lets you do that with SOAP web services. WSDL stands for "Web Services Description Language". Although designed to be flexible enough to describe many types of web services, it is most often used to describe SOAP web services.

A WSDL file is just that: a file. More specifically, it's an XML file. It usually lives on the same server you use to access the SOAP web services it describes, although there's nothing special about it. Later in this chapter, we'll download the WSDL file for the Google API and use it locally. That doesn't mean we're calling Google locally; the WSDL file still describes the remote functions sitting on Google's server.

A WSDL file contains a description of everything involved in calling a SOAP web service:

- The service URL and namespace
- The type of web service (probably function calls using SOAP, although as I mentioned, WSDL is flexible enough to describe a wide variety of web services)
- The list of available functions
- The arguments for each function
- The datatype of each argument
- The return values of each function, and the datatype of each return value

In other words, a WSDL file tells you everything you need to know to be able to call a SOAP web service.

12.6. Introspecting SOAP Web Services with WSDL

Like many things in the web services arena, WSDL has a long and checkered history, full of political strife and intrigue. I will skip over this history entirely, since it bores me to tears. There were other standards that tried to do similar things, but WSDL won, so let's learn how to use it.

The most fundamental thing that WSDL allows you to do is discover the available methods offered by a SOAP server.

Example 12.8. Discovering The Available Methods

```
>>> from SOAPpy import WSDL          ❶
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl')
>>> server = WSDL.Proxy(wsdlFile)    ❷
>>> server.methods.keys()            ❸
[u'getTemp']
```

❶ SOAPpy includes a WSDL parser. At the time of this writing, it was labeled as being in the early stages of development, but I had no problem parsing any of the WSDL files I tried.

❷ To use a WSDL file, you again use a proxy class, `WSDL.Proxy`, which takes a single argument: the WSDL file. Note that in this case you are passing in the URL of a WSDL file stored on the remote server, but the proxy class works just as well with a local copy of the WSDL file. The act of creating the WSDL proxy will download the WSDL file and parse it, so it there are any errors in the WSDL file (or it can't be fetched due to networking problems), you'll know about it immediately.

❸ The WSDL proxy class exposes the available functions as a Python dictionary, `server.methods`. So getting the list of available methods is as simple as calling the dictionary method `keys()`.

Okay, so you know that this SOAP server offers a single method: `getTemp`. But how do you call it? The WSDL proxy object can tell you that too.

Example 12.9. Discovering A Method's Arguments

```
>>> callInfo = server.methods['getTemp']  ❶
>>> callInfo.inparams                     ❷
[<SOAPpy.wstools.WSDLTools.ParameterInfo instance at 0x00CF3AD0>]
>>> callInfo.inparams[0].name             ❸
u'zipcode'
>>> callInfo.inparams[0].type             ❹
(u'http://www.w3.org/2001/XMLSchema', u'string')
```

❶ The `server.methods` dictionary is filled with a SOAPpy−specific structure called `CallInfo`. A `CallInfo` object contains information about one specific function, including the function arguments.

❷ The function arguments are stored in `callInfo.inparams`, which is a Python list of `ParameterInfo` objects that hold information about each parameter.

❸ Each `ParameterInfo` object contains a `name` attribute, which is the argument name. You are not required to know the argument name to call the function through SOAP, but SOAP does support calling functions with named arguments (just like Python), and `WSDL.Proxy` will correctly handle mapping named arguments to the remote function if you choose to use them.

❹ Each parameter is also explicitly typed, using datatypes defined in XML Schema. You saw this in the wire trace in the previous section; the XML Schema namespace was part of the "boilerplate" I told you to ignore. For our purposes here, you may continue to ignore it. The `zipcode` parameter is a string, and if you pass in a Python string to the `WSDL.Proxy` object, it will map it correctly and send it to the server.

WSDL also lets you introspect into a function's return values.

Example 12.10. Discovering A Method's Return Values

```
>>> callInfo.outparams            ❶
[<SOAPpy.wstools.WSDLTools.ParameterInfo instance at 0x00CF3AF8>]
>>> callInfo.outparams[0].name    ❷
u'return'
>>> callInfo.outparams[0].type
(u'http://www.w3.org/2001/XMLSchema', u'float')
```

❶ The adjunct to `callInfo.inparams` for function arguments is `callInfo.outparams` for return value. It is also a list, because functions called through SOAP can return multiple values, just like Python functions.

❷ Each `ParameterInfo` object contains `name` and `type`. This function returns a single value, named `return`, which is a float.

Let's put it all together, and call a SOAP web service through a WSDL proxy.

Example 12.11. Calling A Web Service Through A WSDL Proxy

```
>>> from SOAPpy import WSDL
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl')
>>> server = WSDL.Proxy(wsdlFile)                ❶
>>> server.getTemp('90210')                      ❷
66.0
>>> server.soapproxy.config.dumpSOAPOut = 1      ❸
>>> server.soapproxy.config.dumpSOAPIn = 1
>>> temperature = server.getTemp('90210')
```

```
*** Outgoing SOAP ******************************************************
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">90210</v1>
</ns1:getTemp>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************
*** Incoming SOAP ******************************************************
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">66.0</return>
</ns1:getTempResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************

>>> temperature
66.0
```

1. The configuration is simpler than calling the SOAP service directly, since the WSDL file contains the both service URL and namespace you need to call the service. Creating the `WSDL.Proxy` object downloads the WSDL file, parses it, and configures a `SOAPProxy` object that it uses to call the actual SOAP web service.
2. Once the `WSDL.Proxy` object is created, you can call a function as easily as you did with the `SOAPProxy` object. This is not surprising; the `WSDL.Proxy` is just a wrapper around the `SOAPProxy` with some introspection methods added, so the syntax for calling functions is the same.
3. You can access the `WSDL.Proxy`'s `SOAPProxy` with `server.soapproxy`. This is useful to turning on debugging, so that when you can call functions through the WSDL proxy, its `SOAPProxy` will dump the outgoing and incoming XML documents that are going over the wire.

12.7. Searching Google

Let's finally turn to the sample code that you saw that the beginning of this chapter, which does something more useful and exciting than get the current temperature.

Google provides a SOAP API for programmatically accessing Google search results. To use it, you will need to sign up for Google Web Services.

Procedure 12.4. Signing Up for Google Web Services

1. Go to http://www.google.com/apis/ and create a Google account. This requires only an email address. After you sign up you will receive your Google API license key by email. You will need this key to pass as a parameter whenever you call Google's search functions.
2. Also on http://www.google.com/apis/, download the Google Web APIs developer kit. This includes some sample code in several programming languages (but not Python), and more importantly, it includes the WSDL

file.
3. Decompress the developer kit file and find `GoogleSearch.wsdl`. Copy this file to some permanent location on your local drive. You will need it later in this chapter.

Once you have your developer key and your Google WSDL file in a known place, you can start poking around with Google Web Services.

Example 12.12. Introspecting Google Web Services

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy('/path/to/your/GoogleSearch.wsdl') ❶
>>> server.methods.keys()                                   ❷
[u'doGoogleSearch', u'doGetCachedPage', u'doSpellingSuggestion']
>>> callInfo = server.methods['doGoogleSearch']
>>> for arg in callInfo.inparams:                           ❸
...     print arg.name.ljust(15), arg.type
key             (u'http://www.w3.org/2001/XMLSchema', u'string')
q               (u'http://www.w3.org/2001/XMLSchema', u'string')
start           (u'http://www.w3.org/2001/XMLSchema', u'int')
maxResults      (u'http://www.w3.org/2001/XMLSchema', u'int')
filter          (u'http://www.w3.org/2001/XMLSchema', u'boolean')
restrict        (u'http://www.w3.org/2001/XMLSchema', u'string')
safeSearch      (u'http://www.w3.org/2001/XMLSchema', u'boolean')
lr              (u'http://www.w3.org/2001/XMLSchema', u'string')
ie              (u'http://www.w3.org/2001/XMLSchema', u'string')
oe              (u'http://www.w3.org/2001/XMLSchema', u'string')
```

❶ Getting started with Google web services is easy: just create a `WSDL.Proxy` object and point it at your local copy of Google's WSDL file.

❷ According to the WSDL file, Google offers three functions: `doGoogleSearch`, `doGetCachedPage`, and `doSpellingSuggestion`. These do exactly what they sound like: perform a Google search and return the results programmatically, get access to the cached version of a page from the last time Google saw it, and offer spelling suggestions for commonly misspelled search words.

❸ The `doGoogleSearch` function takes a number of parameters of various types. Note that while the WSDL file can tell you what the arguments are called and what datatype they are, it can't tell you what they mean or how to use them. It could theoretically tell you the acceptable range of values for each parameter, if only specific values were allowed, but Google's WSDL file is not that detailed. `WSDL.Proxy` can't work magic; it can only give you the information provided in the WSDL file.

Here is a brief synopsis of all the parameters to the `doGoogleSearch` function:

- `key` − Your Google API key, which you received when you signed up for Google web services.
- `q` − The search word or phrase you're looking for. The syntax is exactly the same as Google's web form, so if you know any advanced search syntax or tricks, they all work here as well.
- `start` − The index of the result to start on. Like the interactive web version of Google, this function returns 10 results at a time. If you wanted to get the second "page" of results, you would set `start` to 10.
- `maxResults` − The number of results to return. Currently capped at 10, although you can specify fewer if you are only interested in a few results and want to save a little bandwidth.
- `filter` − If `True`, Google will filter out duplicate pages from the results.
- `restrict` − Set this to `country` plus a country code to get results only from a particular country. Example: `countryUK` to search pages in the United Kingdom. You can also specify `linux`, `mac`, or `bsd` to search a Google−defined set of technical sites, or `unclesam` to search sites about the United States

government.

- `safeSearch` − If `True`, Google will filter out porn sites.
- `lr` ("language restrict") − Set this to a language code to get results only in a particular language.
- `ie` and `oe` ("input encoding" and "output encoding") − Deprecated, both must be `utf-8`.

Example 12.13. Searching Google

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy('/path/to/your/GoogleSearch.wsdl')
>>> key = 'YOUR_GOOGLE_API_KEY'
>>> results = server.doGoogleSearch(key, 'mark', 0, 10, False, "",
...     False, "", "utf-8", "utf-8")             ❶
>>> len(results.resultElements)                  ❷
10
>>> results.resultElements[0].URL                ❸
'http://diveintomark.org/'
>>> results.resultElements[0].title
'dive into <b>mark</b>'
```

❶ After setting up the `WSDL.Proxy` object, you can call `server.doGoogleSearch` with all ten parameters. Remember to use your own Google API key that you received when you signed up for Google web services.

❷ There's a lot of information returned, but let's look at the actual search results first. They're stored in `results.resultElements`, and you can access them just like a normal Python list.

❸ Each element in the `resultElements` is an object that has a `URL`, `title`, `snippet`, and other useful attributes. At this point you can use normal Python introspection techniques like **`dir(results.resultElements[0])`** to see the available attributes. Or you can introspect through the WSDL proxy object and look through the function's `outparams`. Each technique will give you the same information.

The `results` object contains more than the actual search results. It also contains information about the search itself, such as how long it took and how many results were found (even though only 10 were returned). The Google web interface shows this information, and you can access it programmatically too.

Example 12.14. Accessing Secondary Information From Google

```
>>> results.searchTime                     ❶
0.224919
>>> results.estimatedTotalResultsCount     ❷
29800000
>>> results.directoryCategories            ❸
[<SOAPpy.Types.structType item at 14367400>:
 {'fullViewableName':
  'Top/Arts/Literature/World_Literature/American/19th_Century/Twain,_Mark',
  'specialEncoding': ''}]
>>> results.directoryCategories[0].fullViewableName
'Top/Arts/Literature/World_Literature/American/19th_Century/Twain,_Mark'
```

❶ This search took 0.224919 seconds. That does not include the time spent sending and receiving the actual SOAP XML documents. It's just the time that Google spent processing your request once it received it.

❷ In total, there were approximately 30 million results. You can access them 10 at a time by changing the `start` parameter and calling `server.doGoogleSearch` again.

❸ For some queries, Google also returns a list of related categories in the Google Directory (http://directory.google.com/). You can append these URLs to http://directory.google.com/ to

construct the link to the directory category page.

12.8. Troubleshooting SOAP Web Services

Of course, the world of SOAP web services is not all happiness and light. Sometimes things go wrong.

As you've seen throughout this chapter, SOAP involves several layers. There's the HTTP layer, since SOAP is sending XML documents to, and receiving XML documents from, an HTTP server. So all the debugging techniques you learned in Chapter 11, *HTTP Web Services* come into play here. You can **`import httplib`** and then set **`httplib.HTTPConnection.debuglevel = 1`** to see the underlying HTTP traffic.

Beyond the underlying HTTP layer, there are a number of things that can go wrong. SOAPpy does an admirable job hiding the SOAP syntax from you, but that also means it can be difficult to determine where the problem is when things don't work.

Here are a few examples of common mistakes that I've made in using SOAP web services, and the errors they generated.

Example 12.15. Calling a Method With an Incorrectly Configured Proxy

```
>>> from SOAPpy import SOAPProxy
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> server = SOAPProxy(url)                                        ❶
>>> server.getTemp('27502')                                        ❷
<Fault SOAP-ENV:Server.BadTargetObjectURI:
Unable to determine object id from call: is the method element namespaced?>
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server.BadTargetObjectURI:
Unable to determine object id from call: is the method element namespaced?>
```

❶ Did you spot the mistake? You're creating a `SOAPProxy` manually, and you've correctly specified the service URL, but you haven't specified the namespace. Since multiple services may be routed through the same service URL, the namespace is essential to determine which service you're trying to talk to, and therefore which method you're really calling.

❷ The server responds by sending a SOAP Fault, which SOAPpy turns into a Python exception of type `SOAPpy.Types.faultType`. All errors returned from any SOAP server will always be SOAP Faults, so you can easily catch this exception. In this case, the human−readable part of the SOAP Fault gives a clue to the problem: the method element is not namespaced, because the original `SOAPProxy` object was not configured with a service namespace.

Misconfiguring the basic elements of the SOAP service is one of the problems that WSDL aims to solve. The WSDL file contains the service URL and namespace, so you can't get it wrong. Of course, there are still other things you can get wrong.

Example 12.16. Calling a Method With the Wrong Arguments

```
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
```

```
>>> server = WSDL.Proxy(wsdlFile)
>>> temperature = server.getTemp(27502)                    ❶
<Fault SOAP-ENV:Server: Exception while handling service request:
services.temperature.TempService.getTemp(int) -- no signature match>   ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server: Exception while handling service request:
services.temperature.TempService.getTemp(int) -- no signature match>
```

❶ Did you spot the mistake? It's a subtle one: you're calling `server.getTemp` with an integer instead of a string. As you saw from introspecting the WSDL file, the `getTemp()` SOAP function takes a single argument, `zipcode`, which must be a string. `WSDL.Proxy` will *not* coerce datatypes for you; you need to pass the exact datatypes that the server expects.

❷ Again, the server returns a SOAP Fault, and the human−readable part of the error gives a clue as to the problem: you're calling a `getTemp` function with an integer value, but there is no function defined with that name that takes an integer. In theory, SOAP allows you to *overload* functions, so you could have two functions in the same SOAP service with the same name and the same number of arguments, but the arguments were of different datatypes. This is why it's important to match the datatypes exactly, and why `WSDL.Proxy` doesn't coerce datatypes for you. If it did, you could end up calling a completely different function! Good luck debugging that one. It's much easier to be picky about datatypes and fail as quickly as possible if you get them wrong.

It's also possible to write Python code that expects a different number of return values than the remote function actually returns.

Example 12.17. Calling a Method and Expecting the Wrong Number of Return Values

```
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
>>> server = WSDL.Proxy(wsdlFile)
>>> (city, temperature) = server.getTemp(27502)  ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unpack non-sequence
```

❶ Did you spot the mistake? `server.getTemp` only returns one value, a float, but you've written code that assumes you're getting two values and trying to assign them to two different variables. Note that this does not fail with a SOAP fault. As far as the remote server is concerned, nothing went wrong at all. The error only occurred *after* the SOAP transaction was complete, `WSDL.Proxy` returned a float, and your local Python interpreter tried to accomodate your request to split it into two different variables. Since the function only returned one value, you get a Python exception trying to split it, not a SOAP Fault.

What about Google's web service? The most common problem I've had with it is that I forget to set the application key properly.

Example 12.18. Calling a Method With An Application−Specific Error

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy(r'/path/to/local/GoogleSearch.wsdl')
>>> results = server.doGoogleSearch('foo', 'mark', 0, 10, False, "", ❶
...     False, "", "utf-8", "utf-8")
```

```
<Fault SOAP-ENV:Server:                                               ❷
 Exception from service object: Invalid authorization key: foo:
 <SOAPpy.Types.structType detail at 14164616>:
 {'stackTrace':
  'com.google.soap.search.GoogleSearchFault: Invalid authorization key: foo
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:220)
   at com.google.soap.search.QueryLimits.validateKey(QueryLimits.java:127)
   at com.google.soap.search.GoogleSearchService.doPublicMethodChecks(
     GoogleSearchService.java:825)
   at com.google.soap.search.GoogleSearchService.doGoogleSearch(
     GoogleSearchService.java:121)
   at sun.reflect.GeneratedMethodAccessor13.invoke(Unknown Source)
   at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
   at java.lang.reflect.Method.invoke(Unknown Source)
   at org.apache.soap.server.RPCRouter.invoke(RPCRouter.java:146)
   at org.apache.soap.providers.RPCJavaProvider.invoke(
     RPCJavaProvider.java:129)
   at org.apache.soap.server.http.RPCRouterServlet.doPost(
     RPCRouterServlet.java:288)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
   at com.google.gse.HttpConnection.runServlet(HttpConnection.java:237)
   at com.google.gse.HttpConnection.run(HttpConnection.java:195)
   at com.google.gse.DispatchQueue$WorkerThread.run(DispatchQueue.java:201)
Caused by: com.google.soap.search.UserKeyInvalidException: Key was of wrong size.
   at com.google.soap.search.UserKey.<init>(UserKey.java:59)
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:217)
   ... 14 more
'}>
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server: Exception from service object:
Invalid authorization key: foo:
<SOAPpy.Types.structType detail at 14164616>:
{'stackTrace':
  'com.google.soap.search.GoogleSearchFault: Invalid authorization key: foo
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:220)
   at com.google.soap.search.QueryLimits.validateKey(QueryLimits.java:127)
   at com.google.soap.search.GoogleSearchService.doPublicMethodChecks(
     GoogleSearchService.java:825)
   at com.google.soap.search.GoogleSearchService.doGoogleSearch(
     GoogleSearchService.java:121)
   at sun.reflect.GeneratedMethodAccessor13.invoke(Unknown Source)
   at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
   at java.lang.reflect.Method.invoke(Unknown Source)
   at org.apache.soap.server.RPCRouter.invoke(RPCRouter.java:146)
   at org.apache.soap.providers.RPCJavaProvider.invoke(
     RPCJavaProvider.java:129)
   at org.apache.soap.server.http.RPCRouterServlet.doPost(
     RPCRouterServlet.java:288)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
   at com.google.gse.HttpConnection.runServlet(HttpConnection.java:237)
   at com.google.gse.HttpConnection.run(HttpConnection.java:195)
```

```
   at com.google.gse.DispatchQueue$WorkerThread.run(DispatchQueue.java:201)
Caused by: com.google.soap.search.UserKeyInvalidException: Key was of wrong size.
   at com.google.soap.search.UserKey.<init>(UserKey.java:59)
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:217)
   ... 14 more
'}>
```

❶ Can you spot the mistake? There's nothing wrong with the calling syntax, or the number of arguments, or the datatypes. The problem is application−specific: the first argument is supposed to be my application key, but `foo` is not a valid Google key.

❷ The Google server responds with a SOAP Fault and an incredibly long error message, which includes a complete Java stack trace. Remember that *all* SOAP errors are signified by SOAP Faults: errors in configuration, errors in function arguments, and application−specific errors like this. Buried in there somewhere is the crucial piece of information: `Invalid authorization key: foo`.

Further Reading on Troubleshooting SOAP

- New developments for SOAPpy (http://www−106.ibm.com/developerworks/webservices/library/ws−pyth17.html) steps through trying to connect to another SOAP service that doesn't quite work as advertised.

12.9. Summary

SOAP web services are very complicated. The specification is very ambitious and tries to cover many different use cases for web services. This chapter has touched on some of the simpler use cases.

Before diving into the next chapter, make sure you're comfortable doing all of these things:

- Connecting to a SOAP server and calling remote methods
- Loading a WSDL file and introspecting remote methods
- Debugging SOAP calls with wire traces
- Troubleshooting common SOAP−related errors

Chapter 13. Unit Testing

13.1. Introduction to Roman numerals

In previous chapters, you "dived in" by immediately looking at code and trying to understand it as quickly as possible. Now that you have some Python under your belt, you're going to step back and look at the steps that happen *before* the code gets written.

In the next few chapters, you're going to write, debug, and optimize a set of utility functions to convert to and from Roman numerals. You saw the mechanics of constructing and validating Roman numerals in Section 7.3, Case Study: Roman Numerals , but now let's step back and consider what it would take to expand that into a two−way utility.

The rules for Roman numerals lead to a number of interesting observations:

1. There is only one correct way to represent a particular number as Roman numerals.
2. The converse is also true: if a string of characters is a valid Roman numeral, it represents only one number (*i.e.* it can only be read one way).
3. There is a limited range of numbers that can be expressed as Roman numerals, specifically `1` through `3999`. (The Romans did have several ways of expressing larger numbers, for instance by having a bar over a numeral to represent that its normal value should be multiplied by `1000`, but you're not going to deal with that. For the purposes of this chapter, let's stipulate that Roman numerals go from `1` to `3999`.)
4. There is no way to represent `0` in Roman numerals. (Amazingly, the ancient Romans had no concept of `0` as a number. Numbers were for counting things you had; how can you count what you don't have?)
5. There is no way to represent negative numbers in Roman numerals.
6. There is no way to represent fractions or non−integer numbers in Roman numerals.

Given all of this, what would you expect out of a set of functions to convert to and from Roman numerals?

`roman.py` requirements

1. `toRoman` should return the Roman numeral representation for all integers `1` to `3999`.
2. `toRoman` should fail when given an integer outside the range `1` to `3999`.
3. `toRoman` should fail when given a non−integer number.
4. `fromRoman` should take a valid Roman numeral and return the number that it represents.
5. `fromRoman` should fail when given an invalid Roman numeral.
6. If you take a number, convert it to Roman numerals, then convert that back to a number, you should end up with the number you started with. So `fromRoman(toRoman(n)) == n` for all `n` in `1..3999`.
7. `toRoman` should always return a Roman numeral using uppercase letters.
8. `fromRoman` should only accept uppercase Roman numerals (*i.e.* it should fail when given lowercase input).

Further reading

- This site (http://www.wilkiecollins.demon.co.uk/roman/front.htm) has more on Roman numerals, including a fascinating history (http://www.wilkiecollins.demon.co.uk/roman/intro.htm) of how Romans and other civilizations really used them (short answer: haphazardly and inconsistently).

13.2. Diving in

Now that you've completely defined the behavior you expect from your conversion functions, you're going to do something a little unexpected: you're going to write a test suite that puts these functions through their paces and makes sure that they behave the way you want them to. You read that right: you're going to write code that tests code that you haven't written yet.

This is called unit testing, since the set of two conversion functions can be written and tested as a unit, separate from any larger program they may become part of later. Python has a framework for unit testing, the appropriately−named `unittest` module.

`unittest` is included with Python 2.1 and later. Python 2.0 users can download it from `pyunit.sourceforge.net` (http://pyunit.sourceforge.net/).

Unit testing is an important part of an overall testing−centric development strategy. If you write unit tests, it is important to write them early (preferably before writing the code that they test), and to keep them updated as code and requirements change. Unit testing is not a replacement for higher−level functional or system testing, but it is important in all phases of development:

- Before writing code, it forces you to detail your requirements in a useful fashion.
- While writing code, it keeps you from over−coding. When all the test cases pass, the function is complete.
- When refactoring code, it assures you that the new version behaves the same way as the old version.
- When maintaining code, it helps you cover your ass when someone comes screaming that your latest change broke their old code. ("But *sir*, all the unit tests passed when I checked it in...")
- When writing code in a team, it increases confidence that the code you're about to commit isn't going to break other peoples' code, because you can run their unittests first. (I've seen this sort of thing in code sprints. A team breaks up the assignment, everybody takes the specs for their task, writes unit tests for it, then shares their unit tests with the rest of the team. That way, nobody goes off too far into developing code that won't play well with others.)

13.3. Introducing `romantest.py`

This is the complete test suite for your Roman numeral conversion functions, which are yet to be written but will eventually be in `roman.py`. It is not immediately obvious how it all fits together; none of these classes or methods reference any of the others. There are good reasons for this, as you'll see shortly.

Example 13.1. `romantest.py`

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
"""Unit test for roman.py"""

import roman
import unittest

class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
```

```
                 (6, 'VI'),
                 (7, 'VII'),
                 (8, 'VIII'),
                 (9, 'IX'),
                 (10, 'X'),
                 (50, 'L'),
                 (100, 'C'),
                 (500, 'D'),
                 (1000, 'M'),
                 (31, 'XXXI'),
                 (148, 'CXLVIII'),
                 (294, 'CCXCIV'),
                 (312, 'CCCXII'),
                 (421, 'CDXXI'),
                 (528, 'DXXVIII'),
                 (621, 'DCXXI'),
                 (782, 'DCCLXXXII'),
                 (870, 'DCCCLXX'),
                 (941, 'CMXLI'),
                 (1043, 'MXLIII'),
                 (1110, 'MCX'),
                 (1226, 'MCCXXVI'),
                 (1301, 'MCCCI'),
                 (1485, 'MCDLXXXV'),
                 (1509, 'MDIX'),
                 (1607, 'MDCVII'),
                 (1754, 'MDCCLIV'),
                 (1832, 'MDCCCXXXII'),
                 (1993, 'MCMXCIII'),
                 (2074, 'MMLXXIV'),
                 (2152, 'MMCLII'),
                 (2212, 'MMCCXII'),
                 (2343, 'MMCCCXLIII'),
                 (2499, 'MMCDXCIX'),
                 (2574, 'MMDLXXIV'),
                 (2646, 'MMDCXLVI'),
                 (2723, 'MMDCCXXIII'),
                 (2892, 'MMDCCCXCII'),
                 (2975, 'MMCMLXXV'),
                 (3051, 'MMMLI'),
                 (3185, 'MMMCLXXXV'),
                 (3250, 'MMMCCL'),
                 (3313, 'MMMCCCXIII'),
                 (3408, 'MMMCDVIII'),
                 (3501, 'MMMDI'),
                 (3610, 'MMMDCX'),
                 (3743, 'MMMDCCXLIII'),
                 (3844, 'MMMDCCCXLIV'),
                 (3888, 'MMMDCCCLXXXVIII'),
                 (3940, 'MMMCMXL'),
                 (3999, 'MMMCMXCIX'))

    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)
            self.assertEqual(numeral, result)

    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)
```

```
class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000)

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, −1)

    def testNonInteger(self):
        """toRoman should fail with non−integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)

class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()
```

Further reading

* The PyUnit home page (http://pyunit.sourceforge.net/) has an in−depth discussion of using the `unittest` framework (http://pyunit.sourceforge.net/pyunit.html), including advanced features not covered in this chapter.
* The PyUnit FAQ (http://pyunit.sourceforge.net/pyunit.html) explains why test cases are stored separately (http://pyunit.sourceforge.net/pyunit.html#WHERE) from the code they test.
* *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes the `unittest` (http://www.python.org/doc/current/lib/module−unittest.html) module.
* ExtremeProgramming.org (http://www.extremeprogramming.org/) discusses why you should write unit tests (http://www.extremeprogramming.org/rules/unittests.html).
* The Portland Pattern Repository (http://www.c2.com/cgi/wiki) has an ongoing discussion of unit tests (http://www.c2.com/cgi/wiki?UnitTests), including a standard definition (http://www.c2.com/cgi/wiki?StandardDefinitionOfUnitTest), why you should code unit tests first (http://www.c2.com/cgi/wiki?CodeUnitTestFirst), and several in−depth case studies (http://www.c2.com/cgi/wiki?UnitTestTrial).

13.4. Testing for success

The most fundamental part of unit testing is constructing individual test cases. A test case answers a single question about the code it is testing.

A test case should be able to...

* ...run completely by itself, without any human input. Unit testing is about automation.
* ...determine by itself whether the function it is testing has passed or failed, without a human interpreting the results.
* ...run in isolation, separate from any other test cases (even if they test the same functions). Each test case is an island.

Given that, let's build the first test case. You have the following requirement:

1. `toRoman` should return the Roman numeral representation for all integers `1` to `3999`.

Example 13.2. `testToRomanKnownValues`

```
class KnownValues(unittest.TestCase):                                  ❶
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
```

```
                (621, 'DCXXI'),
                (782, 'DCCLXXXII'),
                (870, 'DCCCLXX'),
                (941, 'CMXLI'),
                (1043, 'MXLIII'),
                (1110, 'MCX'),
                (1226, 'MCCXXVI'),
                (1301, 'MCCCI'),
                (1485, 'MCDLXXXV'),
                (1509, 'MDIX'),
                (1607, 'MDCVII'),
                (1754, 'MDCCLIV'),
                (1832, 'MDCCCXXXII'),
                (1993, 'MCMXCIII'),
                (2074, 'MMLXXIV'),
                (2152, 'MMCLII'),
                (2212, 'MMCCXII'),
                (2343, 'MMCCCXLIII'),
                (2499, 'MMCDXCIX'),
                (2574, 'MMDLXXIV'),
                (2646, 'MMDCXLVI'),
                (2723, 'MMDCCXXIII'),
                (2892, 'MMDCCCXCII'),
                (2975, 'MMCMLXXV'),
                (3051, 'MMMLI'),
                (3185, 'MMMCLXXXV'),
                (3250, 'MMMCCL'),
                (3313, 'MMMCCCXIII'),
                (3408, 'MMMCDVIII'),
                (3501, 'MMMDI'),
                (3610, 'MMMDCX'),
                (3743, 'MMMDCCXLIII'),
                (3844, 'MMMDCCCXLIV'),
                (3888, 'MMMDCCCLXXXVIII'),
                (3940, 'MMMCMXL'),
                (3999, 'MMMCMXCIX'))                  ❷

    def testToRomanKnownValues(self):                 ❸
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)           ❹ ❺
            self.assertEqual(numeral, result)         ❻
```

❶ To write a test case, first subclass the `TestCase` class of the `unittest` module. This class provides many useful methods which you can use in your test case to test specific conditions.

❷ This is a list of integer/numeral pairs that I verified manually. It includes the lowest ten numbers, the highest number, every number that translates to a single−character Roman numeral, and a random sampling of other valid numbers. The point of a unit test is not to test every possible input, but to test a representative sample.

❸ Every individual test is its own method, which must take no parameters and return no value. If the method exits normally without raising an exception, the test is considered passed; if the method raises an exception, the test is considered failed.

❹ Here you call the actual `toRoman` function. (Well, the function hasn't be written yet, but once it is, this is the line that will call it.) Notice that you have now defined the API for the `toRoman` function: it must take an integer (the number to convert) and return a string (the Roman numeral representation). If the API is different than that, this test is considered failed.

❺ Also notice that you are not trapping any exceptions when you call `toRoman`. This is intentional. `toRoman` shouldn't raise an exception when you call it with valid input, and these input values are all valid. If `toRoman` raises an exception, this test is considered failed.

❻ Assuming the `toRoman` function was defined correctly, called correctly, completed successfully, and returned a value, the last step is to check whether it returned the *right* value. This is a common question, and the `TestCase` class provides a method, `assertEqual`, to check whether two values are equal. If the result returned from `toRoman` (`result`) does not match the known value you were expecting (`numeral`), `assertEqual` will raise an exception and the test will fail. If the two values are equal, `assertEqual` will do nothing. If every value returned from `toRoman` matches the known value you expect, `assertEqual` never raises an exception, so `testToRomanKnownValues` eventually exits normally, which means `toRoman` has passed this test.

13.5. Testing for failure

It is not enough to test that functions succeed when given good input; you must also test that they fail when given bad input. And not just any sort of failure; they must fail in the way you expect.

Remember the other requirements for `toRoman`:

2. `toRoman` should fail when given an integer outside the range `1` to `3999`.
3. `toRoman` should fail when given a non−integer number.

In Python, functions indicate failure by raising exceptions, and the `unittest` module provides methods for testing whether a function raises a particular exception when given bad input.

Example 13.3. Testing bad input to `toRoman`

```
class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000) ❶

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)    ❷

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, −1)

    def testNonInteger(self):
        """toRoman should fail with non−integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)  ❸
```

❶ The `TestCase` class of the `unittest` provides the `assertRaises` method, which takes the following arguments: the exception you're expecting, the function you're testing, and the arguments you're passing that function. (If the function you're testing takes more than one argument, pass them all to `assertRaises`, in order, and it will pass them right along to the function you're testing.) Pay close attention to what you're doing here: instead of calling `toRoman` directly and manually checking that it raises a particular exception (by wrapping it in a `try...except` block), `assertRaises` has encapsulated all of that for us. All you do is give it the exception (`roman.OutOfRangeError`), the function (`toRoman`), and `toRoman`'s arguments (`4000`), and `assertRaises` takes care of calling `toRoman` and checking to make sure that it raises `roman.OutOfRangeError`. (Also note that you're passing the `toRoman` function itself as an argument; you're not calling it, and you're not passing the name of it as a string. Have I mentioned recently how handy it is that everything in Python is an object, including functions and exceptions?)

❷ Along with testing numbers that are too large, you need to test numbers that are too small. Remember, Roman numerals cannot express `0` or negative numbers, so you have a test case for each of those (`testZero` and `testNegative`). In `testZero`, you are testing that `toRoman` raises a `roman.OutOfRangeError` exception when called with `0`; if it does *not* raise a `roman.OutOfRangeError` (either because it returns an actual value, or because it raises some other exception), this test is considered failed.

❸ Requirement #3 specifies that `toRoman` cannot accept a non−integer number, so here you test to make sure that `toRoman` raises a `roman.NotIntegerError` exception when called with `0.5`. If `toRoman` does not raise a `roman.NotIntegerError`, this test is considered failed.

The next two requirements are similar to the first three, except they apply to `fromRoman` instead of `toRoman`:

4. `fromRoman` should take a valid Roman numeral and return the number that it represents.
5. `fromRoman` should fail when given an invalid Roman numeral.

Requirement #4 is handled in the same way as requirement #1, iterating through a sampling of known values and testing each in turn. Requirement #5 is handled in the same way as requirements #2 and #3, by testing a series of bad inputs and making sure `fromRoman` raises the appropriate exception.

Example 13.4. Testing bad input to `fromRoman`

```
class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s) ❶

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
```

❶ Not much new to say about these; the pattern is exactly the same as the one you used to test bad input to `toRoman`. I will briefly note that you have another exception: `roman.InvalidRomanNumeralError`. That makes a total of three custom exceptions that will need to be defined in `roman.py` (along with `roman.OutOfRangeError` and `roman.NotIntegerError`). You'll see how to define these custom exceptions when you actually start writing `roman.py`, later in this chapter.

13.6. Testing for sanity

Often, you will find that a unit of code contains a set of reciprocal functions, usually in the form of conversion functions where one converts A to B and the other converts B to A. In these cases, it is useful to create a "sanity check" to make sure that you can convert A to B and back to A without losing precision, incurring rounding errors, or triggering any other sort of bug.

Consider this requirement:

6. If you take a number, convert it to Roman numerals, then convert that back to a number, you should end up with the number you started with. So `fromRoman(toRoman(n)) == n` for all `n` in `1..3999`.

Example 13.5. Testing `toRoman` against `fromRoman`

```
class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):         ❶ ❷
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result) ❸
```

❶ You've seen the `range` function before, but here it is called with two arguments, which returns a list of integers starting at the first argument (`1`) and counting consecutively up to *but not including* the second argument (`4000`). Thus, `1..3999`, which is the valid range for converting to Roman numerals.

❷ I just wanted to mention in passing that `integer` is not a keyword in Python; here it's just a variable name like any other.

❸ The actual testing logic here is straightforward: take a number (`integer`), convert it to a Roman numeral (`numeral`), then convert it back to a number (`result`) and make sure you end up with the same number you started with. If not, `assertEqual` will raise an exception and the test will immediately be considered failed. If all the numbers match, `assertEqual` will always return silently, the entire `testSanity` method will eventually return silently, and the test will be considered passed.

The last two requirements are different from the others because they seem both arbitrary and trivial:

7. `toRoman` should always return a Roman numeral using uppercase letters.
8. `fromRoman` should only accept uppercase Roman numerals (*i.e.* it should fail when given lowercase input).

In fact, they are somewhat arbitrary. You could, for instance, have stipulated that `fromRoman` accept lowercase and mixed case input. But they are not completely arbitrary; if `toRoman` is always returning uppercase output, then `fromRoman` must at least accept uppercase input, or the "sanity check" (requirement #6) would fail. The fact that it *only* accepts uppercase input is arbitrary, but as any systems integrator will tell you, case always matters, so it's worth specifying the behavior up front. And if it's worth specifying, it's worth testing.

Example 13.6. Testing for case

```
class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())         ❶

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())                   ❷ ❸
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())  ❹
```

1. The most interesting thing about this test case is all the things it doesn't test. It doesn't test that the value returned from `toRoman` is right or even consistent; those questions are answered by separate test cases. You have a whole test case just to test for uppercase−ness. You might be tempted to combine this with the sanity check, since both run through the entire range of values and call `toRoman`.[6] But that would violate one of the fundamental rules: each test case should answer only a single question. Imagine that you combined this case check with the sanity check, and then that test case failed. You would need to do further analysis to figure out which part of the test case failed to determine what the problem was. If you need to analyze the results of your unit testing just to figure out what they mean, it's a sure sign that you've mis−designed your test cases.
2. There's a similar lesson to be learned here: even though "you know" that `toRoman` always returns uppercase, you are explicitly converting its return value to uppercase here to test that `fromRoman` accepts uppercase input. Why? Because the fact that `toRoman` always returns uppercase is an independent requirement. If you changed that requirement so that, for instance, it always returned lowercase, the `testToRomanCase` test case would need to change, but this test case would still work. This was another of the fundamental rules: each test case must be able to work in isolation from any of the others. Every test case is an island.
3. Note that you're not assigning the return value of `fromRoman` to anything. This is legal syntax in Python; if a function returns a value but nobody's listening, Python just throws away the return value. In this case, that's what you want. This test case doesn't test anything about the return value; it just tests that `fromRoman` accepts the uppercase input without raising an exception.
4. This is a complicated line, but it's very similar to what you did in the `ToRomanBadInput` and `FromRomanBadInput` tests. You are testing to make sure that calling a particular function (`roman.fromRoman`) with a particular value (`numeral.lower()`, the lowercase version of the current Roman numeral in the loop) raises a particular exception (`roman.InvalidRomanNumeralError`). If it does (each time through the loop), the test passes; if even one time it does something else (like raises a different exception, or returning a value without raising an exception at all), the test fails.

In the next chapter, you'll see how to write code that passes these tests.

---

[6] "I can resist everything except temptation." −−Oscar Wilde

Chapter 14. Test−First Programming

14.1. `roman.py`, stage 1

Now that the unit tests are complete, it's time to start writing the code that the test cases are attempting to test. You're going to do this in stages, so you can see all the unit tests fail, then watch them pass one by one as you fill in the gaps in `roman.py`.

Example 14.1. `roman1.py`

This file is available in `py/roman/stage1/` in the examples directory.

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass                    ❶
class OutOfRangeError(RomanError): pass              ❷
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass ❸

def toRoman(n):
    """convert integer to Roman numeral"""
    pass                                             ❹

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

❶ This is how you define your own custom exceptions in Python. Exceptions are classes, and you create your own by subclassing existing exceptions. It is strongly recommended (but not required) that you subclass `Exception`, which is the base class that all built−in exceptions inherit from. Here I am defining `RomanError` (inherited from `Exception`) to act as the base class for all my other custom exceptions to follow. This is a matter of style; I could just as easily have inherited each individual exception from the `Exception` class directly.

❷ The `OutOfRangeError` and `NotIntegerError` exceptions will eventually be used by `toRoman` to flag various forms of invalid input, as specified in `ToRomanBadInput`.

❸ The `InvalidRomanNumeralError` exception will eventually be used by `fromRoman` to flag invalid input, as specified in `FromRomanBadInput`.

❹ At this stage, you want to define the API of each of your functions, but you don't want to code them yet, so you stub them out using the Python reserved word `pass`.

Now for the big moment (drum roll please): you're finally going to run the unit test against this stubby little module. At this point, every test case should fail. In fact, if any test case passes in stage 1, you should go back to `romantest.py` and re−evaluate why you coded a test so useless that it passes with do−nothing functions.

Run `romantest1.py` with the `−v` command−line option, which will give more verbose output so you can see exactly what's going on as each test case runs. With any luck, your output should look like this:

Example 14.2. Output of `romantest1.py` against `roman1.py`

```
fromRoman should only accept uppercase input ... ERROR
toRoman should always return uppercase ... ERROR
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... FAIL
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... FAIL
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL

======================================================================
ERROR: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 154, in testFromRomanCase
    roman1.fromRoman(numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
======================================================================
ERROR: toRoman should always return uppercase
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 148, in testToRomanCase
    self.assertEqual(numeral, numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 127, in testRepeatedPairs
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
```

```
======================================================================
FAIL: toRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 93, in testToRomanKnownValues
    self.assertEqual(numeral, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: I != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 116, in testNonInteger
    self.assertRaises(roman1.NotIntegerError, roman1.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError
======================================================================
FAIL: toRoman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 112, in testNegative
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 104, in testTooLarge
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with 0 input                                 ❶
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 108, in testZero
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError                                        ❷
----------------------------------------------------------------------
Ran 12 tests in 0.040s                                                 ❸

FAILED (failures=10, errors=2)                                         ❹
```

❶ Running the script runs `unittest.main()`, which runs each test case, which is to say each method defined in each class within `romantest.py`. For each test case, it prints out the `doc string` of the method and whether that test passed or failed. As expected, none of the test cases passed.

❷ For each failed test case, `unittest` displays the trace information showing exactly what happened. In this case, the call to `assertRaises` (also called `failUnlessRaises`) raised an `AssertionError` because it was expecting `toRoman` to raise an `OutOfRangeError` and it didn't.

❸ After the detail, `unittest` displays a summary of how many tests were performed and how long it took.

❹ Overall, the unit test failed because at least one test case did not pass. When a test case doesn't pass, `unittest` distinguishes between failures and errors. A failure is a call to an `assertXYZ` method, like `assertEqual` or `assertRaises`, that fails because the asserted condition is not true or the expected exception was not raised. An error is any other sort of exception raised in the code you're testing or the unit test case itself. For instance, the `testFromRomanCase` method ("`fromRoman` should only accept uppercase input") was an error, because the call to `numeral.upper()` raised an `AttributeError` exception, because `toRoman` was supposed to return a string but didn't. But `testZero` ("`toRoman` should fail with 0 input") was a failure, because the call to `fromRoman` did not raise the `InvalidRomanNumeral` exception that `assertRaises` was looking for.

14.2. `roman.py`, stage 2

Now that you have the framework of the `roman` module laid out, it's time to start writing code and passing test cases.

Example 14.3. `roman2.py`

This file is available in `py/roman/stage2/` in the examples directory.

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000), ❶
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:      ❷
            result += numeral
            n -= integer
    return result
```

```
def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

❶ romanNumeralMap is a tuple of tuples which defines three things:

1. The character representations of the most basic Roman numerals. Note that this is not just the single−character Roman numerals; you're also defining two−character pairs like CM ("one hundred less than one thousand"); this will make the toRoman code simpler later.
2. The order of the Roman numerals. They are listed in descending value order, from M all the way down to I.
3. The value of each Roman numeral. Each inner tuple is a pair of (*numeral*, *value*).

❷ Here's where your rich data structure pays off, because you don't need any special logic to handle the subtraction rule. To convert to Roman numerals, you simply iterate through romanNumeralMap looking for the largest integer value less than or equal to the input. Once found, you add the Roman numeral representation to the end of the output, subtract the corresponding integer value from the input, lather, rinse, repeat.

Example 14.4. How toRoman works

If you're not clear how toRoman works, add a print statement to the end of the while loop:

```
        while n >= integer:
            result += numeral
            n −= integer
            print 'subtracting', integer, 'from input, adding', numeral, 'to output'
```

```
>>> import roman2
>>> roman2.toRoman(1424)
subtracting 1000 from input, adding M to output
subtracting 400 from input, adding CD to output
subtracting 10 from input, adding X to output
subtracting 10 from input, adding X to output
subtracting 4 from input, adding IV to output
'MCDXXIV'
```

So toRoman appears to work, at least in this manual spot check. But will it pass the unit testing? Well no, not entirely.

Example 14.5. Output of romantest2.py against roman2.py

Remember to run romantest2.py with the −v command−line flag to enable verbose mode.

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok                          ❶
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok               ❷
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non−integer input ... FAIL                    ❸
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL
```

1. `toRoman` does, in fact, always return uppercase, because `romanNumeralMap` defines the Roman numeral representations as uppercase. So this test passes already.
2. Here's the big news: this version of the `toRoman` function passes the known values test. Remember, it's not comprehensive, but it does put the function through its paces with a variety of good inputs, including inputs that produce every single−character Roman numeral, the largest possible input (`3999`), and the input that produces the longest possible Roman numeral (`3888`). At this point, you can be reasonably confident that the function works for any good input value you could throw at it.
3. However, the function does not "work" for bad values; it fails every single bad input test. That makes sense, because you didn't include any checks for bad input. Those test cases look for specific exceptions to be raised (via `assertRaises`), and you're never raising them. You'll do that in the next stage.

Here's the rest of the output of the unit test, listing the details of all the failures. You're down to 10.

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 156, in testFromRomanCase
    roman2.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 127, in testRepeatedPairs
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 116, in testNonInteger
    self.assertRaises(roman2.NotIntegerError, roman2.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError
======================================================================
FAIL: toRoman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 112, in testNegative
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 104, in testTooLarge
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with 0 input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 108, in testZero
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
----------------------------------------------------------------------
Ran 12 tests in 0.320s

FAILED (failures=10)
```

14.3. `roman.py`, stage 3

Now that `toRoman` behaves correctly with good input (integers from `1` to `3999`), it's time to make it behave correctly with bad input (everything else).

Example 14.6. `roman3.py`

This file is available in `py/roman/stage3/` in the examples directory.

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):                                                        ❶
        raise OutOfRangeError, "number out of range (must be 1..3999)"            ❷
    if int(n) <> n:                                                               ❸
        raise NotIntegerError, "non-integers can not be converted"

    result = ""                                                                   ❹
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

❶ This is a nice Pythonic shortcut: multiple comparisons at once. This is equivalent to `if not ((0 < n) and (n < 4000))`, but it's much easier to read. This is the range check, and it should catch inputs that are too large, negative, or zero.

❷ You raise exceptions yourself with the `raise` statement. You can raise any of the built−in exceptions, or you can raise any of your custom exceptions that you've defined. The second parameter, the error message, is optional; if given, it is displayed in the traceback that is printed if the exception is never handled.

❸ This is the non−integer check. Non−integers can not be converted to Roman numerals.

❹ The rest of the function is unchanged.

Example 14.7. Watching `toRoman` handle bad input

```
>>> import roman3
>>> roman3.toRoman(4000)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 27, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
>>> roman3.toRoman(1.5)
```

```
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 29, in toRoman
    raise NotIntegerError, "non-integers can not be converted"
NotIntegerError: non-integers can not be converted
```

Example 14.8. Output of `romantest3.py` against `roman3.py`

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok ❶
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... ok        ❷
toRoman should fail with negative input ... ok           ❸
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

❶ `toRoman` still passes the known values test, which is comforting. All the tests that passed in stage 2 still pass, so the latest code hasn't broken anything.

❷ More exciting is the fact that all of the bad input tests now pass. This test, `testNonInteger`, passes because of the `int(n) <> n` check. When a non−integer is passed to `toRoman`, the `int(n) <> n` check notices it and raises the `NotIntegerError` exception, which is what `testNonInteger` is looking for.

❸ This test, `testNegative`, passes because of the `not (0 < n < 4000)` check, which raises an `OutOfRangeError` exception, which is what `testNegative` is looking for.

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 156, in testFromRomanCase
    roman3.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 127, in testRepeatedPairs
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
```

```
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
----------------------------------------------------------------------
Ran 12 tests in 0.401s

FAILED (failures=6) ❶
```

❶ You're down to 6 failures, and all of them involve `fromRoman`: the known values test, the three separate bad input tests, the case check, and the sanity check. That means that `toRoman` has passed all the tests it can pass by itself. (It's involved in the sanity check, but that also requires that `fromRoman` be written, which it isn't yet.) Which means that you must stop coding `toRoman` now. No tweaking, no twiddling, no extra checks "just in case". Stop. Now. Back away from the keyboard.

The most important thing that comprehensive unit testing can tell you is when to stop coding. When all the unit tests for a function pass, stop coding the function. When all the unit tests for an entire module pass, stop coding the module.

14.4. `roman.py`, stage 4

Now that `toRoman` is done, it's time to start coding `fromRoman`. Thanks to the rich data structure that maps individual Roman numerals to integer values, this is no more difficult than the `toRoman` function.

Example 14.9. `roman4.py`

This file is available in `py/roman/stage4/` in the examples directory.

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass
```

```
#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

# toRoman function omitted for clarity (it hasn't changed)

def fromRoman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral: ❶
            result += integer
            index += len(numeral)
    return result
```

❶ The pattern here is the same as `toRoman`. You iterate through your Roman numeral data structure (a tuple of tuples), and instead of matching the highest integer values as often as possible, you match the "highest" Roman numeral character strings as often as possible.

Example 14.10. How `fromRoman` works

If you're not clear how `fromRoman` works, add a `print` statement to the end of the `while` loop:

```
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
            print 'found', numeral, 'of length', len(numeral), ', adding', integer
```

```
>>> import roman4
>>> roman4.fromRoman('MCMLXXII')
found M , of length 1, adding 1000
found CM , of length 2, adding 900
found L , of length 1, adding 50
found X , of length 1, adding 10
found X , of length 1, adding 10
found I , of length 1, adding 1
found I , of length 1, adding 1
1972
```

Example 14.11. Output of `romantest4.py` against `roman4.py`

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... ok ❶
```

```
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok                     ❷
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

❶ Two pieces of exciting news here. The first is that `fromRoman` works for good input, at least for all the known values you test.

❷ The second is that the sanity check also passed. Combined with the known values tests, you can be reasonably sure that both `toRoman` and `fromRoman` work properly for all possible good values. (This is not guaranteed; it is theoretically possible that `toRoman` has a bug that produces the wrong Roman numeral for some particular set of inputs, *and* that `fromRoman` has a reciprocal bug that produces the same wrong integer values for exactly that set of Roman numerals that `toRoman` generated incorrectly. Depending on your application and your requirements, this possibility may bother you; if so, write more comprehensive test cases until it doesn't bother you.)

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 156, in testFromRomanCase
    roman4.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 127, in testRepeatedPairs
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
----------------------------------------------------------------------
Ran 12 tests in 1.222s

FAILED (failures=4)
```

14.5. `roman.py`, stage 5

Now that `fromRoman` works properly with good input, it's time to fit in the last piece of the puzzle: making it work properly with bad input. That means finding a way to look at a string and determine if it's a valid Roman numeral. This is inherently more difficult than validating numeric input in `toRoman`, but you have a powerful tool at your disposal: regular expressions.

If you're not familiar with regular expressions and didn't read Chapter 7, *Regular Expressions*, now would be a good time.

As you saw in Section 7.3, Case Study: Roman Numerals , there are several simple rules for constructing a Roman numeral, using the letters `M`, `D`, `C`, `L`, `X`, `V`, and `I`. Let's review the rules:

1. Characters are additive. `I` is `1`, `II` is `2`, and `III` is `3`. `VI` is `6` (literally, "`5` and `1`"), `VII` is `7`, and `VIII` is `8`.
2. The tens characters (`I`, `X`, `C`, and `M`) can be repeated up to three times. At `4`, you need to subtract from the next highest fives character. You can't represent `4` as `IIII`; instead, it is represented as `IV` ("`1` less than `5`"). `40` is written as `XL` ("`10` less than `50`"), `41` as `XLI`, `42` as `XLII`, `43` as `XLIII`, and then `44` as `XLIV` ("`10` less than `50`, then `1` less than `5`").
3. Similarly, at `9`, you need to subtract from the next highest tens character: `8` is `VIII`, but `9` is `IX` ("`1` less than `10`"), not `VIIII` (since the `I` character can not be repeated four times). `90` is `XC`, `900` is `CM`.
4. The fives characters can not be repeated. `10` is always represented as `X`, never as `VV`. `100` is always `C`, never `LL`.
5. Roman numerals are always written highest to lowest, and read left to right, so order of characters matters very much. `DC` is `600`; `CD` is a completely different number (`400`, "`100` less than `500`"). `CI` is `101`; `IC` is not even a valid Roman numeral (because you can't subtract `1` directly from `100`; you would need to write it as `XCIX`, "`10` less than `100`, then `1` less than `10`").

Example 14.12. `roman5.py`

This file is available in `py/roman/stage5/` in the examples directory.

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
"""Convert to and from Roman numerals"""
import re

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
```

```
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):
        raise OutOfRangeError, "number out of range (must be 1..3999)"
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$' ❶

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not re.search(romanNumeralPattern, s):                                     ❷
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

❶ This is just a continuation of the pattern you discussed in Section 7.3, Case Study: Roman Numerals . The tens places is either `XC` (`90`), `XL` (`40`), or an optional `L` followed by 0 to 3 optional `X` characters. The ones place is either `IX` (`9`), `IV` (`4`), or an optional `V` followed by 0 to 3 optional `I` characters.

❷ Having encoded all that logic into a regular expression, the code to check for invalid Roman numerals becomes trivial. If `re.search` returns an object, then the regular expression matched and the input is valid; otherwise, the input is invalid.

At this point, you are allowed to be skeptical that that big ugly regular expression could possibly catch all the types of invalid Roman numerals. But don't take my word for it, look at the results:

Example 14.13. Output of `romantest5.py` against `roman5.py`

```
fromRoman should only accept uppercase input ... ok                ❶
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... ok            ❷
fromRoman should fail with repeated pairs of numerals ... ok ❸
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 2.864s
```

```
OK                                                                  (4)
```

1. One thing I didn't mention about regular expressions is that, by default, they are case−sensitive. Since the regular expression `romanNumeralPattern` was expressed in uppercase characters, the `re.search` check will reject any input that isn't completely uppercase. So the uppercase input test passes.
2. More importantly, the bad input tests pass. For instance, the malformed antecedents test checks cases like `MCMC`. As you've seen, this does not match the regular expression, so `fromRoman` raises an `InvalidRomanNumeralError` exception, which is what the malformed antecedents test case is looking for, so the test passes.
3. In fact, all the bad input tests pass. This regular expression catches everything you could think of when you made your test cases.
4. And the anticlimax award of the year goes to the word "`OK`", which is printed by the `unittest` module when all the tests pass.

When all of your tests pass, stop coding.

Chapter 15. Refactoring

15.1. Handling bugs

Despite your best efforts to write comprehensive unit tests, bugs happen. What do I mean by "bug"? A bug is a test case you haven't written yet.

Example 15.1. The bug

```
>>> import roman5
>>> roman5.fromRoman("") ❶
0
```

❶ Remember in the previous section when you kept seeing that an empty string would match the regular expression you were using to check for valid Roman numerals? Well, it turns out that this is still true for the final version of the regular expression. And that's a bug; you want an empty string to raise an `InvalidRomanNumeralError` exception just like any other sequence of characters that don't represent a valid Roman numeral.

After reproducing the bug, and before fixing it, you should write a test case that fails, thus illustrating the bug.

Example 15.2. Testing for the bug (`romantest61.py`)

```
class FromRomanBadInput(unittest.TestCase):

    # previous test cases omitted for clarity (they haven't changed)

    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, "") ❶
```

❶ Pretty simple stuff here. Call `fromRoman` with an empty string and make sure it raises an `InvalidRomanNumeralError` exception. The hard part was finding the bug; now that you know about it, testing for it is the easy part.

Since your code has a bug, and you now have a test case that tests this bug, the test case will fail:

Example 15.3. Output of `romantest61.py` against `roman61.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... FAIL
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

======================================================================
```

```
FAIL: fromRoman should fail with blank string
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage6\romantest61.py", line 137, in testBlank
    self.assertRaises(roman61.InvalidRomanNumeralError, roman61.fromRoman, "")
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
----------------------------------------------------------------------
Ran 13 tests in 2.864s

FAILED (failures=1)
```

Now you can fix the bug.

Example 15.4. Fixing the bug (`roman62.py`)

This file is available in `py/roman/stage6/` in the examples directory.

```
def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s: ❶
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

❶ Only two lines of code are required: an explicit check for an empty string, and a `raise` statement.

Example 15.5. Output of `romantest62.py` against `roman62.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok ❶
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 13 tests in 2.834s

OK ❷
```

❶ The blank string test case now passes, so the bug is fixed.

❷ All the other test cases still pass, which means that this bug fix didn't break anything else. Stop coding.

Coding this way does not make fixing bugs any easier. Simple bugs (like this one) require simple test cases; complex bugs will require complex test cases. In a testing−centric environment, it may *seem* like it takes longer to fix a bug, since you need to articulate in code exactly what the bug is (to write the test case), then fix the bug itself. Then if the test case doesn't pass right away, you need to figure out whether the fix was wrong, or whether the test case itself has a bug in it. However, in the long run, this back−and−forth between test code and code tested pays for itself, because it makes it more likely that bugs are fixed correctly the first time. Also, since you can easily re−run *all* the test cases along with your new one, you are much less likely to break old code when fixing new code. Today's unit test is tomorrow's regression test.

15.2. Handling changing requirements

Despite your best efforts to pin your customers to the ground and extract exact requirements from them on pain of horrible nasty things involving scissors and hot wax, requirements will change. Most customers don't know what they want until they see it, and even if they do, they aren't that good at articulating what they want precisely enough to be useful. And even if they do, they'll want more in the next release anyway. So be prepared to update your test cases as requirements change.

Suppose, for instance, that you wanted to expand the range of the Roman numeral conversion functions. Remember the rule that said that no character could be repeated more than three times? Well, the Romans were willing to make an exception to that rule by having 4 `M` characters in a row to represent `4000`. If you make this change, you'll be able to expand the range of convertible numbers from `1..3999` to `1..4999`. But first, you need to make some changes to the test cases.

Example 15.6. Modifying test cases for new requirements (`romantest71.py`)

This file is available in `py/roman/stage7/` in the examples directory.

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
import roman71
import unittest

class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
```

```
                 (621, 'DCXXI'),
                 (782, 'DCCLXXXII'),
                 (870, 'DCCCLXX'),
                 (941, 'CMXLI'),
                 (1043, 'MXLIII'),
                 (1110, 'MCX'),
                 (1226, 'MCCXXVI'),
                 (1301, 'MCCCI'),
                 (1485, 'MCDLXXXV'),
                 (1509, 'MDIX'),
                 (1607, 'MDCVII'),
                 (1754, 'MDCCLIV'),
                 (1832, 'MDCCCXXXII'),
                 (1993, 'MCMXCIII'),
                 (2074, 'MMLXXIV'),
                 (2152, 'MMCLII'),
                 (2212, 'MMCCXII'),
                 (2343, 'MMCCCXLIII'),
                 (2499, 'MMCDXCIX'),
                 (2574, 'MMDLXXIV'),
                 (2646, 'MMDCXLVI'),
                 (2723, 'MMDCCXXIII'),
                 (2892, 'MMDCCCXCII'),
                 (2975, 'MMCMLXXV'),
                 (3051, 'MMMLI'),
                 (3185, 'MMMCLXXXV'),
                 (3250, 'MMMCCL'),
                 (3313, 'MMMCCCXIII'),
                 (3408, 'MMMCDVIII'),
                 (3501, 'MMMDI'),
                 (3610, 'MMMDCX'),
                 (3743, 'MMMDCCXLIII'),
                 (3844, 'MMMDCCCXLIV'),
                 (3888, 'MMMDCCCLXXXVIII'),
                 (3940, 'MMMCMXL'),
                 (3999, 'MMMCMXCIX'),
                 (4000, 'MMMM'),                                    ❶
                 (4500, 'MMMMD'),
                 (4888, 'MMMMDCCCLXXXVIII'),
                 (4999, 'MMMMCMXCIX'))

    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.toRoman(integer)
            self.assertEqual(numeral, result)

    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)

class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 5000) ❷

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 0)

    def testNegative(self):
```

```
        """toRoman should fail with negative input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, -1)

    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman71.NotIntegerError, roman71.toRoman, 0.5)

class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):      ❸
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, "")

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 5000):                                     ❹
            numeral = roman71.toRoman(integer)
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            roman71.fromRoman(numeral.upper())
            self.assertRaises(roman71.InvalidRomanNumeralError,
                              roman71.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()
```

❶ The existing known values don't change (they're all still reasonable values to test), but you need to add a few more in the `4000` range. Here I've included `4000` (the shortest), `4500` (the second shortest), `4888` (the longest), and `4999` (the largest).

❷ The definition of "large input" has changed. This test used to call `toRoman` with `4000` and expect an error; now that `4000-4999` are good values, you need to bump this up to `5000`.

❸ The definition of "too many repeated numerals" has also changed. This test used to call `fromRoman` with `'MMMM'` and expect an error; now that `MMMM` is considered a valid Roman numeral, you need to

bump this up to `'MMMMM'`.

4. The sanity check and case checks loop through every number in the range, from `1` to `3999`. Since the range has now expanded, these `for` loops need to be updated as well to go up to `4999`.

Now your test cases are up to date with the new requirements, but your code is not, so you expect several of the test cases to fail.

Example 15.7. Output of `romantest71.py` against `roman71.py`

```
fromRoman should only accept uppercase input ... ERROR        (1)
toRoman should always return uppercase ... ERROR
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ERROR (2)
toRoman should give known result with known input ... ERROR   (3)
fromRoman(toRoman(n))==n for all n ... ERROR                  (4)
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

1. Our case checks now fail because they loop from `1` to `4999`, but `toRoman` only accepts numbers from `1` to `3999`, so it will fail as soon the test case hits `4000`.
2. The `fromRoman` known values test will fail as soon as it hits `'MMMM'`, because `fromRoman` still thinks this is an invalid Roman numeral.
3. The `toRoman` known values test will fail as soon as it hits `4000`, because `toRoman` still thinks this is out of range.
4. The sanity check will also fail as soon as it hits `4000`, because `toRoman` still thinks this is out of range.

```
======================================================================
ERROR: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 161, in testFromRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: toRoman should always return uppercase
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 155, in testToRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 102, in testFromRomanKnownValues
    result = roman71.fromRoman(numeral)
  File "roman71.py", line 47, in fromRoman
    raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s
InvalidRomanNumeralError: Invalid Roman numeral: MMMM
======================================================================
```

```
ERROR: toRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 96, in testToRomanKnownValues
    result = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 147, in testSanity
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
----------------------------------------------------------------------
Ran 13 tests in 2.213s

FAILED (errors=5)
```

Now that you have test cases that fail due to the new requirements, you can think about fixing the code to bring it in line with the test cases. (One thing that takes some getting used to when you first start coding unit tests is that the code being tested is never "ahead" of the test cases. While it's behind, you still have some work to do, and as soon as it catches up to the test cases, you stop coding.)

Example 15.8. Coding the new requirements (`roman72.py`)

This file is available in `py/roman/stage7/` in the examples directory.

```
"""Convert to and from Roman numerals"""
import re

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 5000):                                                        ❶
        raise OutOfRangeError, "number out of range (must be 1..4999)"
    if int(n) <> n:
```

```
        raise NotIntegerError, "non-integers can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$' ❷

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

❶ `toRoman` only needs one small change, in the range check. Where you used to check `0 < n < 4000`, you now check `0 < n < 5000`. And you change the error message that you `raise` to reflect the new acceptable range (`1..4999` instead of `1..3999`). You don't need to make any changes to the rest of the function; it handles the new cases already. (It merrily adds `'M'` for each thousand that it finds; given `4000`, it will spit out `'MMMM'`. The only reason it didn't do this before is that you explicitly stopped it with the range check.)

❷ You don't need to make any changes to `fromRoman` at all. The only change is to `romanNumeralPattern`; if you look closely, you'll notice that you added another optional `M` in the first section of the regular expression. This will allow up to 4 `M` characters instead of 3, meaning you will allow the Roman numeral equivalents of `4999` instead of `3999`. The actual `fromRoman` function is completely general; it just looks for repeated Roman numeral characters and adds them up, without caring how many times they repeat. The only reason it didn't handle `'MMMM'` before is that you explicitly stopped it with the regular expression pattern matching.

You may be skeptical that these two small changes are all that you need. Hey, don't take my word for it; see for yourself:

Example 15.9. Output of `romantest72.py` against `roman72.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
```

```
Ran 13 tests in 3.685s

OK ❶
```

❶ All the test cases pass. Stop coding.

Comprehensive unit testing means never having to rely on a programmer who says "Trust me."

15.3. Refactoring

The best thing about comprehensive unit testing is not the feeling you get when all your test cases finally pass, or even the feeling you get when someone else blames you for breaking their code and you can actually *prove* that you didn't. The best thing about unit testing is that it gives you the freedom to refactor mercilessly.

Refactoring is the process of taking working code and making it work better. Usually, "better" means "faster", although it can also mean "using less memory", or "using less disk space", or simply "more elegantly". Whatever it means to you, to your project, in your environment, refactoring is important to the long−term health of any program.

Here, "better" means "faster". Specifically, the `fromRoman` function is slower than it needs to be, because of that big nasty regular expression that you use to validate Roman numerals. It's probably not worth trying to do away with the regular expression altogether (it would be difficult, and it might not end up any faster), but you can speed up the function by precompiling the regular expression.

Example 15.10. Compiling regular expressions

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')               ❶
<SRE_Match object at 01090490>
>>> compiledPattern = re.compile(pattern) ❷
>>> compiledPattern
<SRE_Pattern object at 00F06E28>
>>> dir(compiledPattern)                  ❸
['findall', 'match', 'scanner', 'search', 'split', 'sub', 'subn']
>>> compiledPattern.search('M')           ❹
<SRE_Match object at 01104928>
```

❶ This is the syntax you've seen before: `re.search` takes a regular expression as a string (`pattern`) and a string to match against it (`'M'`). If the pattern matches, the function returns a match object which can be queried to find out exactly what matched and how.

❷ This is the new syntax: `re.compile` takes a regular expression as a string and returns a pattern object. Note there is no string to match here. Compiling a regular expression has nothing to do with matching it against any specific strings (like `'M'`); it only involves the regular expression itself.

❸ The compiled pattern object returned from `re.compile` has several useful−looking functions, including several (like `search` and `sub`) that are available directly in the `re` module.

❹ Calling the compiled pattern object's `search` function with the string `'M'` accomplishes the same thing as calling `re.search` with both the regular expression and the string `'M'`. Only much, much faster. (In fact, the `re.search` function simply compiles the regular expression and calls the resulting pattern object's `search` method for you.)

Whenever you are going to use a regular expression more than once, you should compile it to get a pattern object, then call the methods on the pattern object directly.

Example 15.11. Compiled regular expressions in `roman81.py`

This file is available in `py/roman/stage8/` in the examples directory.

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
# toRoman and rest of module omitted for clarity

romanNumeralPattern = \
    re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$') ❶

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not romanNumeralPattern.search(s):                                    ❷
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

❶ This looks very similar, but in fact a lot has changed. `romanNumeralPattern` is no longer a string; it is a pattern object which was returned from `re.compile`.

❷ That means that you can call methods on `romanNumeralPattern` directly. This will be much, much faster than calling `re.search` every time. The regular expression is compiled once and stored in `romanNumeralPattern` when the module is first imported; then, every time you call `fromRoman`, you can immediately match the input string against the regular expression, without any intermediate steps occurring under the covers.

So how much faster is it to compile regular expressions? See for yourself:

Example 15.12. Output of `romantest81.py` against `roman81.py`

```
.............          ❶
----------------------------------------------------------------------
Ran 13 tests in 3.385s ❷

OK                     ❸
```

❶ Just a note in passing here: this time, I ran the unit test *without* the `-v` option, so instead of the full `doc string` for each test, you only get a dot for each test that passes. (If a test failed, you'd get an `F`, and if it had an error, you'd get an `E`. You'd still get complete tracebacks for each failure and error, so you could track down any problems.)

❷ You ran `13` tests in `3.385` seconds, compared to `3.685` seconds without precompiling the regular expressions. That's an `8%` improvement overall, and remember that most of the time spent during the unit test is spent doing other things. (Separately, I time−tested the regular expressions by themselves, apart from the rest of the unit tests, and found that compiling this regular expression speeds up the `search` by an average of `54%`.) Not bad for such a simple fix.

❸ Oh, and in case you were wondering, precompiling the regular expression didn't break anything, and you

just proved it.

There is one other performance optimization that I want to try. Given the complexity of regular expression syntax, it should come as no surprise that there is frequently more than one way to write the same expression. After some discussion about this module on comp.lang.python (http://groups.google.com/groups?group=comp.lang.python), someone suggested that I try using the `{m,n}` syntax for the optional repeated characters.

Example 15.13. `roman82.py`

This file is available in `py/roman/stage8/` in the examples directory.

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
# rest of program omitted for clarity

#old version
#romanNumeralPattern = \
#   re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$')

#new version
romanNumeralPattern = \
    re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$') ❶
```

❶ You have replaced `M?M?M?M?` with `M{0,4}`. Both mean the same thing: "match 0 to 4 `M` characters". Similarly, `C?C?C?` became `C{0,3}` ("match 0 to 3 `C` characters") and so forth for `X` and `I`.

This form of the regular expression is a little shorter (though not any more readable). The big question is, is it any faster?

Example 15.14. Output of `romantest82.py` against `roman82.py`

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s ❶

OK                     ❷
```

❶ Overall, the unit tests run 2% faster with this form of regular expression. That doesn't sound exciting, but remember that the `search` function is a small part of the overall unit test; most of the time is spent doing other things. (Separately, I time−tested just the regular expressions, and found that the `search` function is `11%` faster with this syntax.) By precompiling the regular expression and rewriting part of it to use this new syntax, you've improved the regular expression performance by over `60%`, and improved the overall performance of the entire unit test by over `10%`.

❷ More important than any performance boost is the fact that the module still works perfectly. This is the freedom I was talking about earlier: the freedom to tweak, change, or rewrite any piece of it and verify that you haven't messed anything up in the process. This is not a license to endlessly tweak your code just for the sake of tweaking it; you had a very specific objective ("make `fromRoman` faster"), and you were able to accomplish that objective without any lingering doubts about whether you introduced new bugs in the process.

One other tweak I would like to make, and then I promise I'll stop refactoring and put this module to bed. As you've seen repeatedly, regular expressions can get pretty hairy and unreadable pretty quickly. I wouldn't like to come back to this module in six months and try to maintain it. Sure, the test cases pass, so I know that it works, but if I can't figure out *how* it works, it's still going to be difficult to add new features, fix new bugs, or otherwise maintain it. As you saw

in Section 7.5, Verbose Regular Expressions , Python provides a way to document your logic line−by−line.

Example 15.15. `roman83.py`

This file is available in `py/roman/stage8/` in the examples directory.

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
# rest of program omitted for clarity

#old version
#romanNumeralPattern = \
#   re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$')

#new version
romanNumeralPattern = re.compile('''
    ^                   # beginning of string
    M{0,4}              # thousands - 0 to 4 M's
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C's),
                        #            or 500-800 (D, followed by 0 to 3 C's)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
                        #        or 50-80 (L, followed by 0 to 3 X's)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
                        #        or 5-8 (V, followed by 0 to 3 I's)
    $                   # end of string
    ''', re.VERBOSE) ❶
```

❶ The `re.compile` function can take an optional second argument, which is a set of one or more flags that control various options about the compiled regular expression. Here you're specifying the `re.VERBOSE` flag, which tells Python that there are in−line comments within the regular expression itself. The comments and all the whitespace around them are *not* considered part of the regular expression; the `re.compile` function simply strips them all out when it compiles the expression. This new, "verbose" version is identical to the old version, but it is infinitely more readable.

Example 15.16. Output of `romantest83.py` against `roman83.py`

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s ❶

OK                     ❷
```

❶ This new, "verbose" version runs at exactly the same speed as the old version. In fact, the compiled pattern objects are the same, since the `re.compile` function strips out all the stuff you added.

❷ This new, "verbose" version passes all the same tests as the old version. Nothing has changed, except that the programmer who comes back to this module in six months stands a fighting chance of understanding how the function works.

15.4. Postscript

A clever reader read the previous section and took it to the next level. The biggest headache (and performance drain) in the program as it is currently written is the regular expression, which is required because you have no other way of breaking down a Roman numeral. But there's only 5000 of them; why don't you just build a lookup table once, then simply read that? This idea gets even better when you realize that you don't need to use regular expressions at all. As

you build the lookup table for converting integers to Roman numerals, you can build the reverse lookup table to convert Roman numerals to integers.

And best of all, he already had a complete set of unit tests. He changed over half the code in the module, but the unit tests stayed the same, so he could prove that his code worked just as well as the original.

Example 15.17. `roman9.py`

This file is available in `py/roman/stage9/` in the examples directory.

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Roman numerals must be less than 5000
MAX_ROMAN_NUMERAL = 4999

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

#Create tables for fast conversion of roman numerals.
#See fillLookupTables() below.
toRomanTable = [ None ]  # Skip an index since Roman numerals have no zero
fromRomanTable = {}

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n <= MAX_ROMAN_NUMERAL):
        raise OutOfRangeError, "number out of range (must be 1..%s)" % MAX_ROMAN_NUMERAL
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"
    return toRomanTable[n]

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, "Input can not be blank"
    if not fromRomanTable.has_key(s):
        raise InvalidRomanNumeralError, "Invalid Roman numeral: %s" % s
    return fromRomanTable[s]

def toRomanDynamic(n):
```

```
    """convert integer to Roman numeral using dynamic programming"""
    result = ""
    for numeral, integer in romanNumeralMap:
        if n >= integer:
            result = numeral
            n -= integer
            break
    if n > 0:
        result += toRomanTable[n]
    return result

def fillLookupTables():
    """compute all the possible roman numerals"""
    #Save the values in two global tables to convert to and from integers.
    for integer in range(1, MAX_ROMAN_NUMERAL + 1):
        romanNumber = toRomanDynamic(integer)
        toRomanTable.append(romanNumber)
        fromRomanTable[romanNumber] = integer

fillLookupTables()
```

So how fast is it?

Example 15.18. Output of `romantest9.py` against `roman9.py`

```
.............
----------------------------------------------------------------------
Ran 13 tests in 0.791s

OK
```

Remember, the best performance you ever got in the original version was 13 tests in 3.315 seconds. Of course, it's not entirely a fair comparison, because this version will take longer to import (when it fills the lookup tables). But since import is only done once, this is negligible in the long run.

The moral of the story?

- Simplicity is a virtue.
- Especially when regular expressions are involved.
- And unit tests can give you the confidence to do large−scale refactoring... even if you didn't write the original code.

15.5. Summary

Unit testing is a powerful concept which, if properly implemented, can both reduce maintenance costs and increase flexibility in any long−term project. It is also important to understand that unit testing is not a panacea, a Magic Problem Solver, or a silver bullet. Writing good test cases is hard, and keeping them up to date takes discipline (especially when customers are screaming for critical bug fixes). Unit testing is not a replacement for other forms of testing, including functional testing, integration testing, and user acceptance testing. But it is feasible, and it does work, and once you've seen it work, you'll wonder how you ever got along without it.

This chapter covered a lot of ground, and much of it wasn't even Python−specific. There are unit testing frameworks for many languages, all of which require you to understand the same basic concepts:

* Designing test cases that are specific, automated, and independent
* Writing test cases *before* the code they are testing
* Writing tests that test good input and check for proper results
* Writing tests that test bad input and check for proper failures
* Writing and updating test cases to illustrate bugs or reflect new requirements
* Refactoring mercilessly to improve performance, scalability, readability, maintainability, or whatever other −ility you're lacking

Additionally, you should be comfortable doing all of the following Python−specific things:

* Subclassing `unittest.TestCase` and writing methods for individual test cases
* Using `assertEqual` to check that a function returns a known value
* Using `assertRaises` to check that a function raises a known exception
* Calling `unittest.main()` in your `if __name__` clause to run all your test cases at once
* Running unit tests in verbose or regular mode

Further reading

* XProgramming.com (http://www.xprogramming.com/) has links to download unit testing frameworks (http://www.xprogramming.com/software.htm) for many different languages.

Chapter 16. Functional Programming

16.1. Diving in

In Chapter 13, *Unit Testing*, you learned about the philosophy of unit testing. In Chapter 14, *Test−First Programming*, you stepped through the implementation of basic unit tests in Python. In Chapter 15, *Refactoring*, you saw how unit testing makes large−scale refactoring easier. This chapter will build on those sample programs, but here we will focus more on advanced Python−specific techniques, rather than on unit testing itself.

The following is a complete Python program that acts as a cheap and simple regression testing framework. It takes unit tests that you've written for individual modules, collects them all into one big test suite, and runs them all at once. I actually use this script as part of the build process for this book; I have unit tests for several of the example programs (not just the `roman.py` module featured in Chapter 13, *Unit Testing*), and the first thing my automated build script does is run this program to make sure all my examples still work. If this regression test fails, the build immediately stops. I don't want to release non−working examples any more than you want to download them and sit around scratching your head and yelling at your monitor and wondering why they don't work.

Example 16.1. `regression.py`

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
"""Regression testing framework

This module will search for scripts in the same directory named
XYZtest.py.  Each such script should be a test suite that tests a
module through PyUnit.  (As of Python 2.1, PyUnit is included in
the standard library as "unittest".)  This script will aggregate all
found test suites into one big test suite and run them all at once.
"""

import sys, os, re, unittest

def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
    load = unittest.defaultTestLoader.loadTestsFromModule
    return unittest.TestSuite(map(load, modules))

if __name__ == "__main__":
    unittest.main(defaultTest="regressionTest")
```

Running this script in the same directory as the rest of the example scripts that come with this book will find all the unit tests, named *`module`*`test.py`, run them as a single test, and pass or fail them all at once.

Example 16.2. Sample output of `regression.py`

```
[you@localhost py]$ python regression.py −v
help should fail with no object ... ok                              (1)
```

```
help should return known result for apihelper ... ok
help should honor collapse argument ... ok
help should honor spacing argument ... ok
buildConnectionString should fail with list input ... ok           ❷
buildConnectionString should fail with string input ... ok
buildConnectionString should fail with tuple input ... ok
buildConnectionString handles empty dictionary ... ok
buildConnectionString returns known result with known input ... ok
fromRoman should only accept uppercase input ... ok                ❸
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
kgp a ref test ... ok
kgp b ref test ... ok
kgp c ref test ... ok
kgp d ref test ... ok
kgp e ref test ... ok
kgp f ref test ... ok
kgp g ref test ... ok

----------------------------------------------------------------------
Ran 29 tests in 2.799s

OK
```

❶ The first 5 tests are from `apihelpertest.py`, which tests the example script from Chapter 4, *The Power Of Introspection*.

❷ The next 5 tests are from `odbchelpertest.py`, which tests the example script from Chapter 2, *Your First Python Program*.

❸ The rest are from `romantest.py`, which you studied in depth in Chapter 13, *Unit Testing*.

16.2. Finding the path

When running Python scripts from the command line, it is sometimes useful to know where the currently running script is located on disk.

This is one of those obscure little tricks that is virtually impossible to figure out on your own, but simple to remember once you see it. The key to it is `sys.argv`. As you saw in Chapter 9, *XML Processing*, this is a list that holds the list of command−line arguments. However, it also holds the name of the running script, exactly as it was called from the command line, and this is enough information to determine its location.

Example 16.3. `fullpath.py`

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
import sys, os
```

```
print 'sys.argv[0] =', sys.argv[0]              ❶
pathname = os.path.dirname(sys.argv[0])         ❷
print 'path =', pathname
print 'full path =', os.path.abspath(pathname)  ❸
```

❶ Regardless of how you run a script, `sys.argv[0]` will always contain the name of the script, exactly as it appears on the command line. This may or may not include any path information, as you'll see shortly.

❷ `os.path.dirname` takes a filename as a string and returns the directory path portion. If the given filename does not include any path information, `os.path.dirname` returns an empty string.

❸ `os.path.abspath` is the key here. It takes a pathname, which can be partial or even blank, and returns a fully qualified pathname.

`os.path.abspath` deserves further explanation. It is very flexible; it can take any kind of pathname.

Example 16.4. Further explanation of `os.path.abspath`

```
>>> import os
>>> os.getcwd()                        ❶
/home/you
>>> os.path.abspath('')                ❷
/home/you
>>> os.path.abspath('.ssh')            ❸
/home/you/.ssh
>>> os.path.abspath('/home/you/.ssh') ❹
/home/you/.ssh
>>> os.path.abspath('.ssh/../foo/')    ❺
/home/you/foo
```

❶ `os.getcwd()` returns the current working directory.

❷ Calling `os.path.abspath` with an empty string returns the current working directory, same as `os.getcwd()`.

❸ Calling `os.path.abspath` with a partial pathname constructs a fully qualified pathname out of it, based on the current working directory.

❹ Calling `os.path.abspath` with a full pathname simply returns it.

❺ `os.path.abspath` also *normalizes* the pathname it returns. Note that this example worked even though I don't actually have a 'foo' directory. `os.path.abspath` never checks your actual disk; this is all just string manipulation.

The pathnames and filenames you pass to `os.path.abspath` do not need to exist.

`os.path.abspath` not only constructs full path names, it also normalizes them. That means that if you are in the `/usr/` directory, `os.path.abspath('bin/../local/bin')` will return `/usr/local/bin`. It normalizes the path by making it as simple as possible. If you just want to normalize a pathname like this without turning it into a full pathname, use `os.path.normpath` instead.

Example 16.5. Sample output from `fullpath.py`

```
[you@localhost py]$ python /home/you/diveintopython/common/py/fullpath.py ❶
sys.argv[0] = /home/you/diveintopython/common/py/fullpath.py
path = /home/you/diveintopython/common/py
full path = /home/you/diveintopython/common/py
[you@localhost diveintopython]$ python common/py/fullpath.py               ❷
sys.argv[0] = common/py/fullpath.py
path = common/py
full path = /home/you/diveintopython/common/py
[you@localhost diveintopython]$ cd common/py
```

```
[you@localhost py]$ python fullpath.py                                       ❸
sys.argv[0] = fullpath.py
path =
full path = /home/you/diveintopython/common/py
```

❶ In the first case, `sys.argv[0]` includes the full path of the script. You can then use the `os.path.dirname` function to strip off the script name and return the full directory name, and `os.path.abspath` simply returns what you give it.

❷ If the script is run by using a partial pathname, `sys.argv[0]` will still contain exactly what appears on the command line. `os.path.dirname` will then give you a partial pathname (relative to the current directory), and `os.path.abspath` will construct a full pathname from the partial pathname.

❸ If the script is run from the current directory without giving any path, `os.path.dirname` will simply return an empty string. Given an empty string, `os.path.abspath` returns the current directory, which is what you want, since the script was run from the current directory.

Like the other functions in the `os` and `os.path` modules, `os.path.abspath` is cross−platform. Your results will look slightly different than my examples if you're running on Windows (which uses backslash as a path separator) or Mac OS (which uses colons), but they'll still work. That's the whole point of the `os` module.

Addendum. One reader was dissatisfied with this solution, and wanted to be able to run all the unit tests in the current directory, not the directory where `regression.py` is located. He suggests this approach instead:

Example 16.6. Running scripts in the current directory

```
import sys, os, re, unittest

def regressionTest():
    path = os.getcwd()       ❶
    sys.path.append(path)    ❷
    files = os.listdir(path) ❸
```

❶ Instead of setting `path` to the directory where the currently running script is located, you set it to the current working directory instead. This will be whatever directory you were in before you ran the script, which is not necessarily the same as the directory the script is in. (Read that sentence a few times until you get it.)

❷ Append this directory to the Python library search path, so that when you dynamically import the unit test modules later, Python can find them. You didn't need to do this when `path` was the directory of the currently running script, because Python always looks in that directory.

❸ The rest of the function is the same.

This technique will allow you to re−use this `regression.py` script on multiple projects. Just put the script in a common directory, then change to the project's directory before running it. All of that project's unit tests will be found and tested, instead of the unit tests in the common directory where `regression.py` is located.

16.3. Filtering lists revisited

You're already familiar with using list comprehensions to filter lists. There is another way to accomplish this same thing, which some people feel is more expressive.

Python has a built−in `filter` function which takes two arguments, a function and a list, and returns a list.[7] The function passed as the first argument to `filter` must itself take one argument, and the list that `filter` returns will contain all the elements from the list passed to `filter` for which the function passed to `filter` returns true.

Got all that? It's not as difficult as it sounds.

Example 16.7. Introducing `filter`

```
>>> def odd(n):                 ❶
...     return n % 2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]
>>> filter(odd, li)             ❷
[1, 3, 5, 9, -3]
>>> [e for e in li if odd(e)]   ❸
>>> filteredList = []
>>> for n in li:                ❹
...     if odd(n):
...         filteredList.append(n)
...
>>> filteredList
[1, 3, 5, 9, -3]
```

❶ `odd` uses the built−in mod function "`%`" to return `True` if `n` is odd and `False` if `n` is even.

❷ `filter` takes two arguments, a function (`odd`) and a list (`li`). It loops through the list and calls `odd` with each element. If `odd` returns a true value (remember, any non−zero value is true in Python), then the element is included in the returned list, otherwise it is filtered out. The result is a list of only the odd numbers from the original list, in the same order as they appeared in the original.

❸ You could accomplish the same thing using list comprehensions, as you saw in Section 4.5, Filtering Lists .

❹ You could also accomplish the same thing with a `for` loop. Depending on your programming background, this may seem more "straightforward", but functions like `filter` are much more expressive. Not only is it easier to write, it's easier to read, too. Reading the `for` loop is like standing too close to a painting; you see all the details, but it may take a few seconds to be able to step back and see the bigger picture: "Oh, you're just filtering the list!"

Example 16.8. `filter` in `regression.py`

```
    files = os.listdir(path)                                ❶
    test = re.compile("test\.py$", re.IGNORECASE)           ❷
    files = filter(test.search, files)                      ❸
```

❶ As you saw in Section 16.2, Finding the path , `path` may contain the full or partial pathname of the directory of the currently running script, or it may contain an empty string if the script is being run from the current directory. Either way, `files` will end up with the names of the files in the same directory as this script you're running.

❷ This is a compiled regular expression. As you saw in Section 15.3, Refactoring , if you're going to use the same regular expression over and over, you should compile it for faster performance. The compiled object has a `search` method which takes a single argument, the string to search. If the regular expression matches the string, the `search` method returns a `Match` object containing information about the regular expression match; otherwise it returns `None`, the Python null value.

❸ For each element in the `files` list, you're going to call the `search` method of the compiled regular expression object, `test`. If the regular expression matches, the method will return a `Match` object, which Python considers to be true, so the element will be included in the list returned by `filter`. If the regular expression does not match, the `search` method will return `None`, which Python considers to be false, so the element will not be included.

Historical note. Versions of Python prior to 2.0 did not have list comprehensions, so you couldn't filter using list comprehensions; the `filter` function was the only game in town. Even with the introduction of list comprehensions in 2.0, some people still prefer the old−style `filter` (and its companion function, `map`, which you'll see later in this chapter). Both techniques work at the moment, so which one you use is a matter of style. There is discussion that `map` and `filter` might be deprecated in a future version of Python, but no decision has been made.

Example 16.9. Filtering using list comprehensions instead

```
files = os.listdir(path)
test = re.compile("test\.py$", re.IGNORECASE)
files = [f for f in files if test.search(f)] ❶
```

❶ This will accomplish exactly the same result as using the `filter` function. Which way is more expressive? That's up to you.

16.4. Mapping lists revisited

You're already familiar with using list comprehensions to map one list into another. There is another way to accomplish the same thing, using the built−in `map` function. It works much the same way as the `filter` function.

Example 16.10. Introducing `map`

```
>>> def double(n):
...     return n*2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]
>>> map(double, li)                       ❶
[2, 4, 6, 10, 18, 20, 512, -6]
>>> [double(n) for n in li]               ❷
[2, 4, 6, 10, 18, 20, 512, -6]
>>> newlist = []
>>> for n in li:                          ❸
...     newlist.append(double(n))
...
>>> newlist
[2, 4, 6, 10, 18, 20, 512, -6]
```

❶ `map` takes a function and a list[8] and returns a new list by calling the function with each element of the list in order. In this case, the function simply multiplies each element by 2.

❷ You could accomplish the same thing with a list comprehension. List comprehensions were first introduced in Python 2.0; `map` has been around forever.

❸ You could, if you insist on thinking like a Visual Basic programmer, use a `for` loop to accomplish the same thing.

Example 16.11. `map` with lists of mixed datatypes

```
>>> li = [5, 'a', (2, 'b')]
>>> map(double, li)                       ❶
[10, 'aa', (2, 'b', 2, 'b')]
```

❶ As a side note, I'd like to point out that `map` works just as well with lists of mixed datatypes, as long as the function you're using correctly handles each type. In this case, the `double` function simply multiplies the given argument by 2, and Python Does The Right Thing depending on the datatype of the argument. For integers, this means actually multiplying it by 2; for strings, it means concatenating the string with itself; for tuples, it means making a new tuple that has all of the elements of the original, then all of the elements of the original again.

All right, enough play time. Let's look at some real code.

Example 16.12. `map` in `regression.py`

```
filenameToModuleName = lambda f: os.path.splitext(f)[0] ❶
moduleNames = map(filenameToModuleName, files)         ❷
```

❶ As you saw in Section 4.7, Using lambda Functions , `lambda` defines an inline function. And as you saw in Example 6.17, Splitting Pathnames , `os.path.splitext` takes a filename and returns a tuple (*`name`*, *`extension`*). So `filenameToModuleName` is a function which will take a filename and strip off the file extension, and return just the name.

❷ Calling `map` takes each filename listed in `files`, passes it to the function `filenameToModuleName`, and returns a list of the return values of each of those function calls. In other words, you strip the file extension off of each filename, and store the list of all those stripped filenames in `moduleNames`.

As you'll see in the rest of the chapter, you can extend this type of data−centric thinking all the way to the final goal, which is to define and execute a single test suite that contains the tests from all of those individual test suites.

16.5. Data−centric programming

By now you're probably scratching your head wondering why this is better than using `for` loops and straight function calls. And that's a perfectly valid question. Mostly, it's a matter of perspective. Using `map` and `filter` forces you to center your thinking around your data.

In this case, you started with no data at all; the first thing you did was get the directory path of the current script, and got a list of files in that directory. That was the bootstrap, and it gave you real data to work with: a list of filenames.

However, you knew you didn't care about all of those files, only the ones that were actually test suites. You had *too much data*, so you needed to `filter` it. How did you know which data to keep? You needed a test to decide, so you defined one and passed it to the `filter` function. In this case you used a regular expression to decide, but the concept would be the same regardless of how you constructed the test.

Now you had the filenames of each of the test suites (and only the test suites, since everything else had been filtered out), but you really wanted module names instead. You had the right amount of data, but it was *in the wrong format*. So you defined a function that would transform a single filename into a module name, and you mapped that function onto the entire list. From one filename, you can get a module name; from a list of filenames, you can get a list of module names.

Instead of `filter`, you could have used a `for` loop with an `if` statement. Instead of `map`, you could have used a `for` loop with a function call. But using `for` loops like that is busywork. At best, it simply wastes time; at worst, it introduces obscure bugs. For instance, you need to figure out how to test for the condition "is this file a test suite?" anyway; that's the application−specific logic, and no language can write that for us. But once you've figured that out, do you really want go to all the trouble of defining a new empty list and writing a `for` loop and an `if` statement and manually calling `append` to add each element to the new list if it passes the condition and then keeping track of which variable holds the new filtered data and which one holds the old unfiltered data? Why not just define the test condition, then let Python do the rest of that work for us?

Oh sure, you could try to be fancy and delete elements in place without creating a new list. But you've been burned by that before. Trying to modify a data structure that you're looping through can be tricky. You delete an element, then loop to the next element, and suddenly you've skipped one. Is Python one of the languages that works that way? How long would it take you to figure it out? Would you remember for certain whether it was safe the next time you tried? Programmers spend so much time and make so many mistakes dealing with purely technical issues like this, and it's all pointless. It doesn't advance your program at all; it's just busywork.

I resisted list comprehensions when I first learned Python, and I resisted `filter` and `map` even longer. I insisted on making my life more difficult, sticking to the familiar way of `for` loops and `if` statements and step−by−step code−centric programming. And my Python programs looked a lot like Visual Basic programs, detailing every step of

every operation in every function. And they had all the same types of little problems and obscure bugs. And it was all pointless.

Let it all go. Busywork code is not important. Data is important. And data is not difficult. It's only data. If you have too much, filter it. If it's not what you want, map it. Focus on the data; leave the busywork behind.

16.6. Dynamically importing modules

OK, enough philosophizing. Let's talk about dynamically importing modules.

First, let's look at how you normally import modules. The `import module` syntax looks in the search path for the named module and imports it by name. You can even import multiple modules at once this way, with a comma−separated list. You did this on the very first line of this chapter's script.

Example 16.13. Importing multiple modules at once

```
import sys, os, re, unittest ❶
```

❶ This imports four modules at once: `sys` (for system functions and access to the command line parameters), `os` (for operating system functions like directory listings), `re` (for regular expressions), and `unittest` (for unit testing).

Now let's do the same thing, but with dynamic imports.

Example 16.14. Importing modules dynamically

```
>>> sys = __import__('sys')           ❶
>>> os = __import__('os')
>>> re = __import__('re')
>>> unittest = __import__('unittest')
>>> sys                               ❷
>>> <module 'sys' (built-in)>
>>> os
>>> <module 'os' from '/usr/local/lib/python2.2/os.pyc'>
```

❶ The built−in `__import__` function accomplishes the same goal as using the `import` statement, but it's an actual function, and it takes a string as an argument.

❷ The variable `sys` is now the `sys` module, just as if you had said `import sys`. The variable `os` is now the `os` module, and so forth.

So `__import__` imports a module, but takes a string argument to do it. In this case the module you imported was just a hard−coded string, but it could just as easily be a variable, or the result of a function call. And the variable that you assign the module to doesn't need to match the module name, either. You could import a series of modules and assign them to a list.

Example 16.15. Importing a list of modules dynamically

```
>>> moduleNames = ['sys', 'os', 're', 'unittest'] ❶
>>> moduleNames
['sys', 'os', 're', 'unittest']
>>> modules = map(__import__, moduleNames)        ❷
>>> modules                                       ❸
[<module 'sys' (built-in)>,
```

```
<module 'os' from 'c:\Python22\lib\os.pyc'>,
<module 're' from 'c:\Python22\lib\re.pyc'>,
<module 'unittest' from 'c:\Python22\lib\unittest.pyc'>]
>>> modules[0].version                                  ❹
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'
>>> import sys
>>> sys.version
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'
```

❶ `moduleNames` is just a list of strings. Nothing fancy, except that the strings happen to be names of modules that you could import, if you wanted to.

❷ Surprise, you wanted to import them, and you did, by mapping the `__import__` function onto the list. Remember, this takes each element of the list (`moduleNames`) and calls the function (`__import__`) over and over, once with each element of the list, builds a list of the return values, and returns the result.

❸ So now from a list of strings, you've created a list of actual modules. (Your paths may be different, depending on your operating system, where you installed Python, the phase of the moon, etc.)

❹ To drive home the point that these are real modules, let's look at some module attributes. Remember, `modules[0]` *is* the `sys` module, so `modules[0].version` *is* `sys.version`. All the other attributes and methods of these modules are also available. There's nothing magic about the `import` statement, and there's nothing magic about modules. Modules are objects. Everything is an object.

Now you should be able to put this all together and figure out what most of this chapter's code sample is doing.

16.7. Putting it all together

You've learned enough now to deconstruct the first seven lines of this chapter's code sample: reading a directory and importing selected modules within it.

Example 16.16. The `regressionTest` function

```
def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
load = unittest.defaultTestLoader.loadTestsFromModule
return unittest.TestSuite(map(load, modules))
```

Let's look at it line by line, interactively. Assume that the current directory is `c:\diveintopython\py`, which contains the examples that come with this book, including this chapter's script. As you saw in Section 16.2, Finding the path , the script directory will end up in the `path` variable, so let's start hard−code that and go from there.

Example 16.17. Step 1: Get all the files

```
>>> import sys, os, re, unittest
>>> path = r'c:\diveintopython\py'
>>> files = os.listdir(path)
>>> files ❶
```

```
['BaseHTMLProcessor.py', 'LICENSE.txt', 'apihelper.py', 'apihelpertest.py',
'argecho.py', 'autosize.py', 'builddialectexamples.py', 'dialect.py',
'fileinfo.py', 'fullpath.py', 'kgptest.py', 'makerealworddoc.py',
'odbchelper.py', 'odbchelpertest.py', 'parsephone.py', 'piglatin.py',
'plural.py', 'pluraltest.py', 'pyfontify.py', 'regression.py', 'roman.py', 'romantest.py',
'uncurly.py', 'unicode2koi8r.py', 'urllister.py', 'kgp', 'plural', 'roman',
'colorize.py']
```

❶ `files` is a list of all the files and directories in the script's directory. (If you've been running some of the examples already, you may also see some `.pyc` files in there as well.)

Example 16.18. Step 2: Filter to find the files you care about

```
>>> test = re.compile("test\.py$", re.IGNORECASE)           ❶
>>> files = filter(test.search, files)                      ❷
>>> files                                                   ❸
['apihelpertest.py', 'kgptest.py', 'odbchelpertest.py', 'pluraltest.py', 'romantest.py']
```

❶ This regular expression will match any string that ends with `test.py`. Note that you need to escape the period, since a period in a regular expression usually means "match any single character", but you actually want to match a literal period instead.

❷ The compiled regular expression acts like a function, so you can use it to filter the large list of files and directories, to find the ones that match the regular expression.

❸ And you're left with the list of unit testing scripts, because they were the only ones named `SOMETHINGtest.py`.

Example 16.19. Step 3: Map filenames to module names

```
>>> filenameToModuleName = lambda f: os.path.splitext(f)[0] ❶
>>> filenameToModuleName('romantest.py')                    ❷
'romantest'
>>> filenameToModuleName('odchelpertest.py')
'odbchelpertest'
>>> moduleNames = map(filenameToModuleName, files)          ❸
>>> moduleNames                                             ❹
['apihelpertest', 'kgptest', 'odbchelpertest', 'pluraltest', 'romantest']
```

❶ As you saw in Section 4.7, Using lambda Functions , `lambda` is a quick−and−dirty way of creating an inline, one−line function. This one takes a filename with an extension and returns just the filename part, using the standard library function `os.path.splitext` that you saw in Example 6.17, Splitting Pathnames .

❷ `filenameToModuleName` is a function. There's nothing magic about `lambda` functions as opposed to regular functions that you define with a `def` statement. You can call the `filenameToModuleName` function like any other, and it does just what you wanted it to do: strips the file extension off of its argument.

❸ Now you can apply this function to each file in the list of unit test files, using `map`.

❹ And the result is just what you wanted: a list of modules, as strings.

Example 16.20. Step 4: Mapping module names to modules

```
>>> modules = map(__import__, moduleNames)                  ❶
>>> modules                                                 ❷
[<module 'apihelpertest' from 'apihelpertest.py'>,
<module 'kgptest' from 'kgptest.py'>,
<module 'odbchelpertest' from 'odbchelpertest.py'>,
```

```
<module 'pluraltest' from 'pluraltest.py'>,
<module 'romantest' from 'romantest.py'>]
>>> modules[-1]                                                    ❸
<module 'romantest' from 'romantest.py'>
```

❶ As you saw in Section 16.6, Dynamically importing modules , you can use a combination of `map` and `__import__` to map a list of module names (as strings) into actual modules (which you can call or access like any other module).

❷ `modules` is now a list of modules, fully accessible like any other module.

❸ The last module in the list *is* the `romantest` module, just as if you had said `import romantest`.

Example 16.21. Step 5: Loading the modules into a test suite

```
>>> load = unittest.defaultTestLoader.loadTestsFromModule
>>> map(load, modules)                              ❶
[<unittest.TestSuite tests=[
  <unittest.TestSuite tests=[<apihelpertest.BadInput testMethod=testNoObject>]>,
  <unittest.TestSuite tests=[<apihelpertest.KnownValues testMethod=testApiHelper>]>,
  <unittest.TestSuite tests=[
    <apihelpertest.ParamChecks testMethod=testCollapse>,
    <apihelpertest.ParamChecks testMethod=testSpacing>]>,
    ...
  ]
]
>>> unittest.TestSuite(map(load, modules)) ❷
```

❶ These are real module objects. Not only can you access them like any other module, instantiate classes and call functions, you can also introspect into the module to figure out which classes and functions it has in the first place. That's what the `loadTestsFromModule` method does: it introspects into each module and returns a `unittest.TestSuite` object for each module. Each `TestSuite` object actually contains a list of `TestSuite` objects, one for each `TestCase` class in your module, and each of those `TestSuite` objects contains a list of tests, one for each test method in your module.

❷ Finally, you wrap the list of `TestSuite` objects into one big test suite. The `unittest` module has no problem traversing this tree of nested test suites within test suites; eventually it gets down to an individual test method and executes it, verifies that it passes or fails, and moves on to the next one.

This introspection process is what the `unittest` module usually does for us. Remember that magic−looking `unittest.main()` function that our individual test modules called to kick the whole thing off? `unittest.main()` actually creates an instance of `unittest.TestProgram`, which in turn creates an instance of a `unittest.defaultTestLoader` and loads it up with the module that called it. (How does it get a reference to the module that called it if you don't give it one? By using the equally−magic `__import__('__main__')` command, which dynamically imports the currently−running module. I could write a book on all the tricks and techniques used in the `unittest` module, but then I'd never finish this one.)

Example 16.22. Step 6: Telling `unittest` to use your test suite

```
if __name__ == "__main__":
    unittest.main(defaultTest="regressionTest") ❶
```

❶ Instead of letting the `unittest` module do all its magic for us, you've done most of it yourself. You've created a function (`regressionTest`) that imports the modules yourself, calls `unittest.defaultTestLoader` yourself, and wraps it all up in a test suite. Now all you need to do is tell `unittest` that, instead of looking for tests and building a test suite in the usual way, it should just call the `regressionTest` function,

which returns a ready−to−use `TestSuite`.

16.8. Summary

The `regression.py` program and its output should now make perfect sense.

You should now feel comfortable doing all of these things:

- Manipulating path information from the command line.
- Filtering lists using `filter` instead of list comprehensions.
- Mapping lists using `map` instead of list comprehensions.
- Dynamically importing modules.

---

[7] Technically, the second argument to `filter` can be any sequence, including lists, tuples, and custom classes that act like lists by defining the `__getitem__` special method. If possible, `filter` will return the same datatype as you give it, so filtering a list returns a list, but filtering a tuple returns a tuple.

[8] Again, I should point out that `map` can take a list, a tuple, or any object that acts like a sequence. See previous footnote about `filter`.

Chapter 17. Dynamic functions

17.1. Diving in

I want to talk about plural nouns. Also, functions that return other functions, advanced regular expressions, and generators. Generators are new in Python 2.3. But first, let's talk about how to make plural nouns.

If you haven't read Chapter 7, *Regular Expressions*, now would be a good time. This chapter assumes you understand the basics of regular expressions, and quickly descends into more advanced uses.

English is a schizophrenic language that borrows from a lot of other languages, and the rules for making singular nouns into plural nouns are varied and complex. There are rules, and then there are exceptions to those rules, and then there are exceptions to the exceptions.

If you grew up in an English−speaking country or learned English in a formal school setting, you're probably familiar with the basic rules:

1. If a word ends in S, X, or Z, add ES. "Bass" becomes "basses", "fax" becomes "faxes", and "waltz" becomes "waltzes".
2. If a word ends in a noisy H, add ES; if it ends in a silent H, just add S. What's a noisy H? One that gets combined with other letters to make a sound that you can hear. So "coach" becomes "coaches" and "rash" becomes "rashes", because you can hear the CH and SH sounds when you say them. But "cheetah" becomes "cheetahs", because the H is silent.
3. If a word ends in Y that sounds like I, change the Y to IES; if the Y is combined with a vowel to sound like something else, just add S. So "vacancy" becomes "vacancies", but "day" becomes "days".
4. If all else fails, just add S and hope for the best.

(I know, there are a lot of exceptions. "Man" becomes "men" and "woman" becomes "women", but "human" becomes "humans". "Mouse" becomes "mice" and "louse" becomes "lice", but "house" becomes "houses". "Knife" becomes "knives" and "wife" becomes "wives", but "lowlife" becomes "lowlifes". And don't even get me started on words that are their own plural, like "sheep", "deer", and "haiku".)

Other languages are, of course, completely different.

Let's design a module that pluralizes nouns. Start with just English nouns, and just these four rules, but keep in mind that you'll inevitably need to add more rules, and you may eventually need to add more languages.

17.2. `plural.py`, stage 1

So you're looking at words, which at least in English are strings of characters. And you have rules that say you need to find different combinations of characters, and then do different things to them. This sounds like a job for regular expressions.

Example 17.1. `plural1.py`

```
import re

def plural(noun):
    if re.search('[sxz]$', noun):             ❶
        return re.sub('$', 'es', noun)        ❷
    elif re.search('[^aeioudgkprt]h$', noun):
```

```
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

❶ OK, this is a regular expression, but it uses a syntax you didn't see in Chapter 7, *Regular Expressions*. The square brackets mean "match exactly one of these characters". So `[sxz]` means "`s`, or `x`, or `z`", but only one of them. The `$` should be familiar; it matches the end of string. So you're checking to see if `noun` ends with `s`, `x`, or `z`.

❷ This `re.sub` function performs regular expression−based string substitutions. Let's look at it in more detail.

Example 17.2. Introducing `re.sub`

```
>>> import re
>>> re.search('[abc]', 'Mark')   ❶
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.sub('[abc]', 'o', 'Mark') ❷
'Mork'
>>> re.sub('[abc]', 'o', 'rock') ❸
'rook'
>>> re.sub('[abc]', 'o', 'caps') ❹
'oops'
```

❶ Does the string `Mark` contain `a`, `b`, or `c`? Yes, it contains `a`.

❷ OK, now find `a`, `b`, or `c`, and replace it with `o`. `Mark` becomes `Mork`.

❸ The same function turns `rock` into `rook`.

❹ You might think this would turn `caps` into `oaps`, but it doesn't. `re.sub` replaces *all* of the matches, not just the first one. So this regular expression turns `caps` into `oops`, because both the `c` and the `a` get turned into `o`.

Example 17.3. Back to `plural1.py`

```
import re

def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun)          ❶
    elif re.search('[^aeioudgkprt]h$', noun): ❷
        return re.sub('$', 'es', noun)          ❸
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

❶ Back to the `plural` function. What are you doing? You're replacing the end of string with `es`. In other words, adding `es` to the string. You could accomplish the same thing with string concatenation, for example `noun + 'es'`, but I'm using regular expressions for everything, for consistency, for reasons that will become clear later in the chapter.

❷ Look closely, this is another new variation. The `^` as the first character inside the square brackets means something special: negation. `[^abc]` means "any single character *except* `a`, `b`, or `c`". So `[^aeioudgkprt]` means any character except `a`, `e`, `i`, `o`, `u`, `d`, `g`, `k`, `p`, `r`, or `t`. Then that character needs to be followed by `h`, followed by end of string. You're looking for words that end in H where the H can be heard.

❸ Same pattern here: match words that end in Y, where the character before the Y is *not* `a`, `e`, `i`, `o`, or `u`. You're looking for words that end in Y that sounds like I.

Example 17.4. More on negation regular expressions

```
>>> import re
>>> re.search('[^aeiou]y$', 'vacancy') ❶
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.search('[^aeiou]y$', 'boy')     ❷
>>>
>>> re.search('[^aeiou]y$', 'day')
>>>
>>> re.search('[^aeiou]y$', 'pita')    ❸
>>>
```

❶ `vacancy` matches this regular expression, because it ends in `cy`, and `c` is not `a`, `e`, `i`, `o`, or `u`.

❷ `boy` does not match, because it ends in `oy`, and you specifically said that the character before the `y` could not be `o`. `day` does not match, because it ends in `ay`.

❸ `pita` does not match, because it does not end in `y`.

Example 17.5. More on `re.sub`

```
>>> re.sub('y$', 'ies', 'vacancy')           ❶
'vacancies'
>>> re.sub('y$', 'ies', 'agency')
'agencies'
>>> re.sub('([^aeiou])y$', r'\1ies', 'vacancy') ❷
'vacancies'
```

❶ This regular expression turns `vacancy` into `vacancies` and `agency` into `agencies`, which is what you wanted. Note that it would also turn `boy` into `boies`, but that will never happen in the function because you did that `re.search` first to find out whether you should do this `re.sub`.

❷ Just in passing, I want to point out that it is possible to combine these two regular expressions (one to find out if the rule applies, and another to actually apply it) into a single regular expression. Here's what that would look like. Most of it should look familiar: you're using a remembered group, which you learned in Section 7.6, Case study: Parsing Phone Numbers , to remember the character before the `y`. Then in the substitution string, you use a new syntax, `\1`, which means "hey, that first group you remembered? put it here". In this case, you remember the `c` before the `y`, and then when you do the substitution, you substitute `c` in place of `c`, and `ies` in place of `y`. (If you have more than one remembered group, you can use `\2` and `\3` and so on.)

Regular expression substitutions are extremely powerful, and the `\1` syntax makes them even more powerful. But combining the entire operation into one regular expression is also much harder to read, and it doesn't directly map to the way you first described the pluralizing rules. You originally laid out rules like "if the word ends in S, X, or Z, then add ES". And if you look at this function, you have two lines of code that say "if the word ends in S, X, or Z, then add ES". It doesn't get much more direct than that.

17.3. `plural.py`, stage 2

Now you're going to add a level of abstraction. You started by defining a list of rules: if this, then do that, otherwise go to the next rule. Let's temporarily complicate part of the program so you can simplify another part.

Example 17.6. `plural2.py`

```
import re

def match_sxz(noun):
```

```
    return re.search('[sxz]$', noun)

def apply_sxz(noun):
    return re.sub('$', 'es', noun)

def match_h(noun):
    return re.search('[^aeioudgkprt]h$', noun)

def apply_h(noun):
    return re.sub('$', 'es', noun)

def match_y(noun):
    return re.search('[^aeiou]y$', noun)

def apply_y(noun):
    return re.sub('y$', 'ies', noun)

def match_default(noun):
    return 1

def apply_default(noun):
    return noun + 's'

rules = ((match_sxz, apply_sxz),
         (match_h, apply_h),
         (match_y, apply_y),
         (match_default, apply_default)
         )                                       ❶

def plural(noun):
    for matchesRule, applyRule in rules:         ❷
        if matchesRule(noun):                    ❸
            return applyRule(noun)               ❹
```

❶ This version looks more complicated (it's certainly longer), but it does exactly the same thing: try to match four different rules, in order, and apply the appropriate regular expression when a match is found. The difference is that each individual match and apply rule is defined in its own function, and the functions are then listed in this `rules` variable, which is a tuple of tuples.

❷ Using a `for` loop, you can pull out the match and apply rules two at a time (one match, one apply) from the `rules` tuple. On the first iteration of the `for` loop, `matchesRule` will get `match_sxz`, and `applyRule` will get `apply_sxz`. On the second iteration (assuming you get that far), `matchesRule` will be assigned `match_h`, and `applyRule` will be assigned `apply_h`.

❸ Remember that everything in Python is an object, including functions. `rules` contains actual functions; not names of functions, but actual functions. When they get assigned in the `for` loop, then `matchesRule` and `applyRule` are actual functions that you can call. So on the first iteration of the `for` loop, this is equivalent to calling `matches_sxz(noun)`.

❹ On the first iteration of the `for` loop, this is equivalent to calling `apply_sxz(noun)`, and so forth.

If this additional level of abstraction is confusing, try unrolling the function to see the equivalence. This `for` loop is equivalent to the following:

Example 17.7. Unrolling the `plural` function

```
def plural(noun):
    if match_sxz(noun):
        return apply_sxz(noun)
    if match_h(noun):
        return apply_h(noun)
```

```
    if match_y(noun):
        return apply_y(noun)
    if match_default(noun):
        return apply_default(noun)
```

The benefit here is that that `plural` function is now simplified. It takes a list of rules, defined elsewhere, and iterates through them in a generic fashion. Get a match rule; does it match? Then call the apply rule. The rules could be defined anywhere, in any way. The `plural` function doesn't care.

Now, was adding this level of abstraction worth it? Well, not yet. Let's consider what it would take to add a new rule to the function. Well, in the previous example, it would require adding an `if` statement to the `plural` function. In this example, it would require adding two functions, `match_foo` and `apply_foo`, and then updating the `rules` list to specify where in the order the new match and apply functions should be called relative to the other rules.

This is really just a stepping stone to the next section. Let's move on.

17.4. `plural.py`, stage 3

Defining separate named functions for each match and apply rule isn't really necessary. You never call them directly; you define them in the `rules` list and call them through there. Let's streamline the rules definition by anonymizing those functions.

Example 17.8. `plural3.py`

```
import re

rules = \
  (
    (
     lambda word: re.search('[sxz]$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeioudgkprt]h$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeiou]y$', word),
     lambda word: re.sub('y$', 'ies', word)
    ),
    (
     lambda word: re.search('$', word),
     lambda word: re.sub('$', 's', word)
    )
   )                                           ❶

def plural(noun):
    for matchesRule, applyRule in rules:       ❷
        if matchesRule(noun):
            return applyRule(noun)
```

❶ This is the same set of rules as you defined in stage 2. The only difference is that instead of defining named functions like `match_sxz` and `apply_sxz`, you have "inlined" those function definitions directly into the `rules` list itself, using lambda functions.

❷ Note that the `plural` function hasn't changed at all. It iterates through a set of rule functions, checks

the first rule, and if it returns a true value, calls the second rule and returns the value. Same as above, word for word. The only difference is that the rule functions were defined inline, anonymously, using lambda functions. But the `plural` function doesn't care how they were defined; it just gets a list of rules and blindly works through them.

Now to add a new rule, all you need to do is define the functions directly in the `rules` list itself: one match rule, and one apply rule. But defining the rule functions inline like this makes it very clear that you have some unnecessary duplication here. You have four pairs of functions, and they all follow the same pattern. The match function is a single call to `re.search`, and the apply function is a single call to `re.sub`. Let's factor out these similarities.

17.5. `plural.py`, stage 4

Let's factor out the duplication in the code so that defining new rules can be easier.

Example 17.9. `plural4.py`

```
import re

def buildMatchAndApplyFunctions((pattern, search, replace)):
    matchFunction = lambda word: re.search(pattern, word)      ❶
    applyFunction = lambda word: re.sub(search, replace, word) ❷
    return (matchFunction, applyFunction)                      ❸
```

❶ `buildMatchAndApplyFunctions` is a function that builds other functions dynamically. It takes `pattern`, `search` and `replace` (actually it takes a tuple, but more on that in a minute), and you can build the match function using the `lambda` syntax to be a function that takes one parameter (`word`) and calls `re.search` with the `pattern` that was passed to the `buildMatchAndApplyFunctions` function, and the `word` that was passed to the match function you're building. Whoa.

❷ Building the apply function works the same way. The apply function is a function that takes one parameter, and calls `re.sub` with the `search` and `replace` parameters that were passed to the `buildMatchAndApplyFunctions` function, and the `word` that was passed to the apply function you're building. This technique of using the values of outside parameters within a dynamic function is called *closures*. You're essentially defining constants within the apply function you're building: it takes one parameter (`word`), but it then acts on that plus two other values (`search` and `replace`) which were set when you defined the apply function.

❸ Finally, the `buildMatchAndApplyFunctions` function returns a tuple of two values: the two functions you just created. The constants you defined within those functions (`pattern` within `matchFunction`, and `search` and `replace` within `applyFunction`) stay with those functions, even after you return from `buildMatchAndApplyFunctions`. That's insanely cool.

If this is incredibly confusing (and it should be, this is weird stuff), it may become clearer when you see how to use it.

Example 17.10. `plural4.py` continued

```
patterns = \
  (
    ('[sxz]$', '$', 'es'),
    ('[^aeioudgkprt]h$', '$', 'es'),
    ('(qu|[^aeiou])y$', 'y$', 'ies'),
    ('$', '$', 's')
  )                                            ❶
rules = map(buildMatchAndApplyFunctions, patterns)  ❷
```

❶ Our pluralization rules are now defined as a series of strings (not functions). The first string is the regular expression that you would use in `re.search` to see if this rule matches; the second and third are the search and replace expressions you would use in `re.sub` to actually apply the rule to turn a noun into its plural.

❷ This line is magic. It takes the list of strings in `patterns` and turns them into a list of functions. How? By mapping the strings to the `buildMatchAndApplyFunctions` function, which just happens to take three strings as parameters and return a tuple of two functions. This means that `rules` ends up being exactly the same as the previous example: a list of tuples, where each tuple is a pair of functions, where the first function is the match function that calls `re.search`, and the second function is the apply function that calls `re.sub`.

I swear I am not making this up: `rules` ends up with exactly the same list of functions as the previous example. Unroll the `rules` definition, and you'll get this:

Example 17.11. Unrolling the rules definition

```
rules = \
  (
    (
     lambda word: re.search('[sxz]$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeioudgkprt]h$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeiou]y$', word),
     lambda word: re.sub('y$', 'ies', word)
    ),
    (
     lambda word: re.search('$', word),
     lambda word: re.sub('$', 's', word)
    )
   )
```

Example 17.12. `plural4.py`, finishing up

```
def plural(noun):
    for matchesRule, applyRule in rules:              ❶
        if matchesRule(noun):
            return applyRule(noun)
```

❶ Since the `rules` list is the same as the previous example, it should come as no surprise that the `plural` function hasn't changed. Remember, it's completely generic; it takes a list of rule functions and calls them in order. It doesn't care how the rules are defined. In stage 2, they were defined as seperate named functions. In stage 3, they were defined as anonymous `lambda` functions. Now in stage 4, they are built dynamically by mapping the `buildMatchAndApplyFunctions` function onto a list of raw strings. Doesn't matter; the `plural` function still works the same way.

Just in case that wasn't mind−blowing enough, I must confess that there was a subtlety in the definition of `buildMatchAndApplyFunctions` that I skipped over. Let's go back and take another look.

Example 17.13. Another look at `buildMatchAndApplyFunctions`

```
def buildMatchAndApplyFunctions((pattern, search, replace)):   ❶
```

❶ Notice the double parentheses? This function doesn't actually take three parameters; it actually takes one parameter, a tuple of three elements. But the tuple is expanded when the function is called, and the three elements of the tuple are each assigned to different variables: `pattern`, `search`, and `replace`. Confused yet? Let's see it in action.

Example 17.14. Expanding tuples when calling functions

```
>>> def foo((a, b, c)):
...     print c
...     print b
...     print a
>>> parameters = ('apple', 'bear', 'catnap')
>>> foo(parameters) ❶
catnap
bear
apple
```

❶ The proper way to call the function `foo` is with a tuple of three elements. When the function is called, the elements are assigned to different local variables within `foo`.

Now let's go back and see why this auto−tuple−expansion trick was necessary. `patterns` was a list of tuples, and each tuple had three elements. When you called `map(buildMatchAndApplyFunctions, patterns)`, that means that `buildMatchAndApplyFunctions` is *not* getting called with three parameters. Using `map` to map a single list onto a function always calls the function with a single parameter: each element of the list. In the case of `patterns`, each element of the list is a tuple, so `buildMatchAndApplyFunctions` always gets called with the tuple, and you use the auto−tuple−expansion trick in the definition of `buildMatchAndApplyFunctions` to assign the elements of that tuple to named variables that you can work with.

17.6. `plural.py`, stage 5

You've factored out all the duplicate code and added enough abstractions so that the pluralization rules are defined in a list of strings. The next logical step is to take these strings and put them in a separate file, where they can be maintained separately from the code that uses them.

First, let's create a text file that contains the rules you want. No fancy data structures, just space− (or tab−)delimited strings in three columns. You'll call it `rules.en`; "en" stands for English. These are the rules for pluralizing English nouns. You could add other rule files for other languages later.

Example 17.15. `rules.en`

```
[sxz]$                  $               es
[^aeioudgkprt]h$        $               es
[^aeiou]y$              y$              ies
$                       $               s
```

Now let's see how you can use this rules file.

Example 17.16. `plural5.py`

```
import re
import string

def buildRule((pattern, search, replace)):
    return lambda word: re.search(pattern, word) and re.sub(search, replace, word) ❶
```

```
def plural(noun, language='en'):                             ❷
    lines = file('rules.%s' % language).readlines()          ❸
    patterns = map(string.split, lines)                      ❹
    rules = map(buildRule, patterns)                         ❺
    for rule in rules:
        result = rule(noun)                                  ❻
        if result: return result
```

❶ You're still using the closures technique here (building a function dynamically that uses variables defined outside the function), but now you've combined the separate match and apply functions into one. (The reason for this change will become clear in the next section.) This will let you accomplish the same thing as having two functions, but you'll need to call it differently, as you'll see in a minute.

❷ Our `plural` function now takes an optional second parameter, `language`, which defaults to `en`.

❸ You use the `language` parameter to construct a filename, then open the file and read the contents into a list. If `language` is `en`, then you'll open the `rules.en` file, read the entire thing, break it up by carriage returns, and return a list. Each line of the file will be one element in the list.

❹ As you saw, each line in the file really has three values, but they're separated by whitespace (tabs or spaces, it makes no difference). Mapping the `string.split` function onto this list will create a new list where each element is a tuple of three strings. So a line like `[sxz]$ $ es` will be broken up into the tuple `('[sxz]$', '$', 'es')`. This means that `patterns` will end up as a list of tuples, just like you hard−coded it in stage 4.

❺ If `patterns` is a list of tuples, then `rules` will be a list of the functions created dynamically by each call to `buildRule`. Calling `buildRule(('[sxz]$', '$', 'es'))` returns a function that takes a single parameter, `word`. When this returned function is called, it will execute `re.search('[sxz]$', word) and re.sub('$', 'es', word)`.

❻ Because you're now building a combined match−and−apply function, you need to call it differently. Just call the function, and if it returns something, then that's the plural; if it returns nothing (`None`), then the rule didn't match and you need to try another rule.

So the improvement here is that you've completely separated the pluralization rules into an external file. Not only can the file be maintained separately from the code, but you've set up a naming scheme where the same `plural` function can use different rule files, based on the `language` parameter.

The downside here is that you're reading that file every time you call the `plural` function. I thought I could get through this entire book without using the phrase "left as an exercise for the reader", but here you go: building a caching mechanism for the language−specific rule files that auto−refreshes itself if the rule files change between calls *is left as an exercise for the reader*. Have fun.

17.7. `plural.py`, stage 6

Now you're ready to talk about generators.

Example 17.17. `plural6.py`

```
import re

def rules(language):
    for line in file('rules.%s' % language):
        pattern, search, replace = line.split()
        yield lambda word: re.search(pattern, word) and re.sub(search, replace, word)

def plural(noun, language='en'):
```

```
    for applyRule in rules(language):
        result = applyRule(noun)
        if result: return result
```

This uses a technique called generators, which I'm not even going to try to explain until you look at a simpler example first.

Example 17.18. Introducing generators

```
>>> def make_counter(x):
...     print 'entering make_counter'
...     while 1:
...         yield x               ❶
...         print 'incrementing x'
...         x = x + 1
...
>>> counter = make_counter(2) ❷
>>> counter                   ❸
<generator object at 0x001C9C10>
>>> counter.next()            ❹
entering make_counter
2
>>> counter.next()            ❺
incrementing x
3
>>> counter.next()            ❻
incrementing x
4
```

❶ The presence of the `yield` keyword in `make_counter` means that this is not a normal function. It is a special kind of function which generates values one at a time. You can think of it as a resumable function. Calling it will return a generator that can be used to generate successive values of `x`.

❷ To create an instance of the `make_counter` generator, just call it like any other function. Note that this does not actually execute the function code. You can tell this because the first line of `make_counter` is a `print` statement, but nothing has been printed yet.

❸ The `make_counter` function returns a generator object.

❹ The first time you call the `next()` method on the generator object, it executes the code in `make_counter` up to the first `yield` statement, and then returns the value that was yielded. In this case, that will be `2`, because you originally created the generator by calling `make_counter(2)`.

❺ Repeatedly calling `next()` on the generator object *resumes where you left off* and continues until you hit the next `yield` statement. The next line of code waiting to be executed is the `print` statement that prints `incrementing x`, and then after that the `x = x + 1` statement that actually increments it. Then you loop through the `while` loop again, and the first thing you do is `yield x`, which returns the current value of `x` (now 3).

❻ The second time you call `counter.next()`, you do all the same things again, but this time `x` is now `4`. And so forth. Since `make_counter` sets up an infinite loop, you could theoretically do this forever, and it would just keep incrementing `x` and spitting out values. But let's look at more productive uses of generators instead.

Example 17.19. Using generators instead of recursion

```
def fibonacci(max):
    a, b = 0, 1       ❶
    while a < max:
        yield a       ❷
        a, b = b, a+b ❸
```

❶ The Fibonacci sequence is a sequence of numbers where each number is the sum of the two numbers before it. It starts with `0` and `1`, goes up slowly at first, then more and more rapidly. To start the sequence, you need two variables: `a` starts at `0`, and `b` starts at `1`.

❷ `a` is the current number in the sequence, so yield it.

❸ `b` is the next number in the sequence, so assign that to `a`, but also calculate the next value (`a+b`) and assign that to `b` for later use. Note that this happens in parallel; if `a` is `3` and `b` is `5`, then `a, b = b, a+b` will set `a` to `5` (the previous value of `b`) and `b` to `8` (the sum of the previous values of `a` and `b`).

So you have a function that spits out successive Fibonacci numbers. Sure, you could do that with recursion, but this way is easier to read. Also, it works well with `for` loops.

Example 17.20. Generators in `for` loops

```
>>> for n in fibonacci(1000): ❶
...     print n,              ❷
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

❶ You can use a generator like `fibonacci` in a `for` loop directly. The `for` loop will create the generator object and successively call the `next()` method to get values to assign to the `for` loop index variable (`n`).

❷ Each time through the `for` loop, `n` gets a new value from the `yield` statement in `fibonacci`, and all you do is print it out. Once `fibonacci` runs out of numbers (`a` gets bigger than `max`, which in this case is `1000`), then the `for` loop exits gracefully.

OK, let's go back to the `plural` function and see how you're using this.

Example 17.21. Generators that generate dynamic functions

```
def rules(language):
    for line in file('rules.%s' % language):                                   ❶
        pattern, search, replace = line.split()                                ❷
        yield lambda word: re.search(pattern, word) and re.sub(search, replace, word) ❸

def plural(noun, language='en'):
    for applyRule in rules(language):  ❹
        result = applyRule(noun)
        if result: return result
```

❶ `for line in file(...)` is a common idiom for reading lines from a file, one line at a time. It works because *`file` actually returns a generator* whose `next()` method returns the next line of the file. That is so insanely cool, I wet myself just thinking about it.

❷ No magic here. Remember that the lines of the rules file have three values separated by whitespace, so `line.split()` returns a tuple of 3 values, and you assign those values to 3 local variables.

❸ *And then you yield.* What do you yield? A function, built dynamically with `lambda`, that is actually a closure (it uses the local variables `pattern`, `search`, and `replace` as constants). In other words, `rules` is a generator that spits out rule functions.

❹ Since `rules` is a generator, you can use it directly in a `for` loop. The first time through the `for` loop, you will call the `rules` function, which will open the rules file, read the first line out of it, dynamically build a function that matches and applies the first rule defined in the rules file, and yields the dynamically built function. The second time through the `for` loop, you will pick up where you left off in `rules` (which was in the middle of the `for line in file(...)` loop), read the second line of the rules file, dynamically build another function that matches and applies the second rule defined in the rules file, and yields it. And so forth.

What have you gained over stage 5? In stage 5, you read the entire rules file and built a list of all the possible rules before you even tried the first one. Now with generators, you can do everything lazily: you open the first and read the first rule and create a function to try it, but if that works you don't ever read the rest of the file or create any other functions.

Further reading

- PEP 255 (http://www.python.org/peps/pep−0255.html) defines generators.
- Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/) has many more examples of generators (http://www.google.com/search?q=generators+cookbook+site:aspn.activestate.com).

17.8. Summary

You talked about several different advanced techniques in this chapter. Not all of them are appropriate for every situation.

You should now be comfortable with all of these techniques:

- Performing string substitution with regular expressions.
- Treating functions as objects, storing them in lists, assigning them to variables, and calling them through those variables.
- Building dynamic functions with `lambda`.
- Building closures, dynamic functions that contain surrounding variables as constants.
- Building generators, resumable functions that perform incremental logic and return different values each time you call them.

Adding abstractions, building functions dynamically, building closures, and using generators can all make your code simpler, more readable, and more flexible. But they can also end up making it more difficult to debug later. It's up to you to find the right balance between simplicity and power.

Chapter 18. Performance Tuning

Performance tuning is a many−splendored thing. Just because Python is an interpreted language doesn't mean you shouldn't worry about code optimization. But don't worry about it *too* much.

18.1. Diving in

There are so many pitfalls involved in optimizing your code, it's hard to know where to start.

Let's start here: *are you sure you need to do it at all?* Is your code really so bad? Is it worth the time to tune it? Over the lifetime of your application, how much time is going to be spent running that code, compared to the time spent waiting for a remote database server, or waiting for user input?

Second, *are you sure you're done coding?* Premature optimization is like spreading frosting on a half−baked cake. You spend hours or days (or more) optimizing your code for performance, only to discover it doesn't do what you need it to do. That's time down the drain.

This is not to say that code optimization is worthless, but you need to look at the whole system and decide whether it's the best use of your time. Every minute you spend optimizing code is a minute you're not spending adding new features, or writing documentation, or playing with your kids, or writing unit tests.

Oh yes, unit tests. It should go without saying that you need a complete set of unit tests before you begin performance tuning. The last thing you need is to introduce new bugs while fiddling with your algorithms.

With these caveats in place, let's look at some techniques for optimizing Python code. The code in question is an implementation of the Soundex algorithm. Soundex was a method used in the early 20th century for categorizing surnames in the United States census. It grouped similar−sounding names together, so even if a name was misspelled, researchers had a chance of finding it. Soundex is still used today for much the same reason, although of course we use computerized database servers now. Most database servers include a Soundex function.

There are several subtle variations of the Soundex algorithm. This is the one used in this chapter:

1. Keep the first letter of the name as−is.
2. Convert the remaining letters to digits, according to a specific table:
    - B, F, P, and V become 1.
    - C, G, J, K, Q, S, X, and Z become 2.
    - D and T become 3.
    - L becomes 4.
    - M and N become 5.
    - R becomes 6.
    - All other letters become 9.
3. Remove consecutive duplicates.
4. Remove all 9s altogether.
5. If the result is shorter than four characters (the first letter plus three digits), pad the result with trailing zeros.
6. if the result is longer than four characters, discard everything after the fourth character.

For example, my name, `Pilgrim`, becomes P942695. That has no consecutive duplicates, so nothing to do there. Then you remove the 9s, leaving P4265. That's too long, so you discard the excess character, leaving P426.

Another example: `Woo` becomes W99, which becomes W9, which becomes W, which gets padded with zeros to become W000.

Here's a first attempt at a Soundex function:

Example 18.1. `soundex/stage1/soundex1a.py`

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
import string, re

charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
                 "S": "2",
                 "T": "3",
                 "U": "9",
                 "V": "1",
                 "W": "9",
                 "X": "2",
                 "Y": "9",
                 "Z": "2"}

def soundex(source):
    "convert string to Soundex equivalent"

    # Soundex requirements:
    # source string must be at least 1 character
    # and must consist entirely of letters
    allChars = string.uppercase + string.lowercase
    if not re.search('^[%s]+$' % allChars, source):
        return "0000"

    # Soundex algorithm:
    # 1. make first character uppercase
    source = source[0].upper() + source[1:]

    # 2. translate all other characters to Soundex digits
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]

    # 3. remove consecutive duplicates
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
```

```
    # 4. remove all "9"s
    digits3 = re.sub('9', '', digits2)

    # 5. pad end with "0"s to 4 characters
    while len(digits3) < 4:
        digits3 += "0"

    # 6. return first 4 characters
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

Further Reading on Soundex

- Soundexing and Genealogy (http://www.avotaynu.com/soundex.html) gives a chronology of the evolution of the Soundex and its regional variations.

18.2. Using the `timeit` Module

The most important thing you need to know about optimizing Python code is that you shouldn't write your own timing function.

Timing short pieces of code is incredibly complex. How much processor time is your computer devoting to running this code? Are there things running in the background? Are you sure? Every modern computer has background processes running, some all the time, some intermittently. Cron jobs fire off at consistent intervals; background services occasionally "wake up" to do useful things like check for new mail, connect to instant messaging servers, check for application updates, scan for viruses, check whether a disk has been inserted into your CD drive in the last 100 nanoseconds, and so on. Before you start your timing tests, turn everything off and disconnect from the network. Then turn off all the things you forgot to turn off the first time, then turn off the service that's incessantly checking whether the network has come back yet, then ...

And then there's the matter of the variations introduced by the timing framework itself. Does the Python interpreter cache method name lookups? Does it cache code block compilations? Regular expressions? Will your code have side effects if run more than once? Don't forget that you're dealing with small fractions of a second, so small mistakes in your timing framework will irreparably skew your results.

The Python community has a saying: "Python comes with batteries included." Don't write your own timing framework. Python 2.3 comes with a perfectly good one called `timeit`.

Example 18.2. Introducing `timeit`

If you have not already done so, you can download this and other examples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) used in this book.

```
>>> import timeit
>>> t = timeit.Timer("soundex.soundex('Pilgrim')",
...     "import soundex")   ❶
>>> t.timeit()              ❷
```

```
8.21683733547
>>> t.repeat(3, 2000000)      ❸
[16.48319309109, 16.46128984923, 16.44203948912]
```

❶ The `timeit` module defines one class, `Timer`, which takes two arguments. Both arguments are strings. The first argument is the statement you wish to time; in this case, you are timing a call to the Soundex function within the `soundex` with an argument of `'Pilgrim'`. The second argument to the `Timer` class is the import statement that sets up the environment for the statement. Internally, `timeit` sets up an isolated virtual environment, manually executes the setup statement (importing the `soundex` module), then manually compiles and executes the timed statement (calling the Soundex function).

❷ Once you have the `Timer` object, the easiest thing to do is call `timeit()`, which calls your function 1 million times and returns the number of seconds it took to do it.

❸ The other major method of the `Timer` object is `repeat()`, which takes two optional arguments. The first argument is the number of times to repeat the entire test, and the second argument is the number of times to call the timed statement within each test. Both arguments are optional, and they default to `3` and `1000000` respectively. The `repeat()` method returns a list of the times each test cycle took, in seconds.

You can use the `timeit` module on the command line to test an existing Python program, without modifying the code. See http://docs.python.org/lib/node396.html for documentation on the command−line flags.

Note that `repeat()` returns a list of times. The times will almost never be identical, due to slight variations in how much processor time the Python interpreter is getting (and those pesky background processes that you can't get rid of). Your first thought might be to say "Let's take the average and call that The True Number."

In fact, that's almost certainly wrong. The tests that took longer didn't take longer because of variations in your code or in the Python interpreter; they took longer because of those pesky background processes, or other factors outside of the Python interpreter that you can't fully eliminate. If the different timing results differ by more than a few percent, you still have too much variability to trust the results. Otherwise, take the minimum time and discard the rest.

Python has a handy `min` function that takes a list and returns the smallest value:

```
>>> min(t.repeat(3, 1000000))
8.22203948912
```

The `timeit` module only works if you already know what piece of code you need to optimize. If you have a larger Python program and don't know where your performance problems are, check out the `hotshot` module. (http://docs.python.org/lib/module−hotshot.html)

18.3. Optimizing Regular Expressions

The first thing the Soundex function checks is whether the input is a non−empty string of letters. What's the best way to do this?

If you answered "regular expressions", go sit in the corner and contemplate your bad instincts. Regular expressions are almost never the right answer; they should be avoided whenever possible. Not only for performance reasons, but simply because they're difficult to debug and maintain. Also for performance reasons.

This code fragment from `soundex/stage1/soundex1a.py` checks whether the function argument `source` is a word made entirely of letters, with at least one letter (not the empty string):

```
    allChars = string.uppercase + string.lowercase
    if not re.search('^[%s]+$' % allChars, source):
        return "0000"
```

How does `soundex1a.py` perform? For convenience, the `__main__` section of the script contains this code that calls the `timeit` module, sets up a timing test with three different names, tests each name three times, and displays the minimum time for each:

```
if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

So how does `soundex1a.py` perform with this regular expression?

```
C:\samples\soundex\stage1>python soundex1a.py
Woo             W000 19.3356647283
Pilgrim         P426 24.0772053431
Flingjingwaller F452 35.0463220884
```

As you might expect, the algorithm takes significantly longer when called with longer names. There will be a few things we can do to narrow that gap (make the function take less relative time for longer input), but the nature of the algorithm dictates that it will never run in constant time.

The other thing to keep in mind is that we are testing a representative sample of names. `Woo` is a kind of trivial case, in that it gets shorted down to a single letter and then padded with zeros. `Pilgrim` is a normal case, of average length and a mixture of significant and ignored letters. `Flingjingwaller` is extraordinarily long and contains consecutive duplicates. Other tests might also be helpful, but this hits a good range of different cases.

So what about that regular expression? Well, it's inefficient. Since the expression is testing for ranges of characters (`A-Z` in uppercase, and `a-z` in lowercase), we can use a shorthand regular expression syntax. Here is `soundex/stage1/soundex1b.py`:

```
    if not re.search('^[A-Za-z]+$', source):
        return "0000"
```

`timeit` says `soundex1b.py` is slightly faster than `soundex1a.py`, but nothing to get terribly excited about:

```
C:\samples\soundex\stage1>python soundex1b.py
Woo             W000 17.1361133887
Pilgrim         P426 21.8201693232
Flingjingwaller F452 32.7262294509
```

We saw in Section 15.3, Refactoring that regular expressions can be compiled and reused for faster results. Since this regular expression never changes across function calls, we can compile it once and use the compiled version. Here is `soundex/stage1/soundex1c.py`:

```
isOnlyChars = re.compile('^[A-Za-z]+$').search
def soundex(source):
    if not isOnlyChars(source):
        return "0000"
```

Using a compiled regular expression in `soundex1c.py` is significantly faster:

```
C:\samples\soundex\stage1>python soundex1c.py
Woo             W000 14.5348347346
Pilgrim         P426 19.2784703084
Flingjingwaller F452 30.0893873383
```

But is this the wrong path? The logic here is simple: the input `source` needs to be non−empty, and it needs to be composed entirely of letters. Wouldn't it be faster to write a loop checking each character, and do away with regular expressions altogether?

Here is `soundex/stage1/soundex1d.py`:

```
    if not source:
        return "0000"
    for c in source:
        if not ('A' <= c <= 'Z') and not ('a' <= c <= 'z'):
            return "0000"
```

It turns out that this technique in `soundex1d.py` is *not* faster than using a compiled regular expression (although it is faster than using a non−compiled regular expression):

```
C:\samples\soundex\stage1>python soundex1d.py
Woo             W000 15.4065058548
Pilgrim         P426 22.2753567842
Flingjingwaller F452 37.5845122774
```

Why isn't `soundex1d.py` faster? The answer lies in the interpreted nature of Python. The regular expression engine is written in C, and compiled to run natively on your computer. On the other hand, this loop is written in Python, and runs through the Python interpreter. Even though the loop is relatively simple, it's not simple enough to make up for the overhead of being interpreted. Regular expressions are never the right answer... except when they are.

It turns out that Python offers an obscure string method. You can be excused for not knowing about it, since it's never been mentioned in this book. The method is called `isalpha()`, and it checks whether a string contains only letters.

This is `soundex/stage1/soundex1e.py`:

```
    if (not source) and (not source.isalpha()):
        return "0000"
```

How much did we gain by using this specific method in `soundex1e.py`? Quite a bit.

```
C:\samples\soundex\stage1>python soundex1e.py
Woo             W000 13.5069504644
Pilgrim         P426 18.2199394057
Flingjingwaller F452 28.9975225902
```

Example 18.3. Best Result So Far: `soundex/stage1/soundex1e.py`

```
import string, re

charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
```

```
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
                 "S": "2",
                 "T": "3",
                 "U": "9",
                 "V": "1",
                 "W": "9",
                 "X": "2",
                 "Y": "9",
                 "Z": "2"}

def soundex(source):
    if (not source) and (not source.isalpha()):
        return "0000"
    source = source[0].upper() + source[1:]
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

18.4. Optimizing Dictionary Lookups

The second step of the Soundex algorithm is to convert characters to digits in a specific pattern. What's the best way to do this?

The most obvious solution is to define a dictionary with individual characters as keys and their corresponding digits as values, and do dictionary lookups on each character. This is what we have in `soundex/stage1/soundex1c.py` (the current best result so far):

```
charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
```

```
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
                 "S": "2",
                 "T": "3",
                 "U": "9",
                 "V": "1",
                 "W": "9",
                 "X": "2",
                 "Y": "9",
                 "Z": "2"}

def soundex(source):
    # ... input check omitted for brevity ...
    source = source[0].upper() + source[1:]
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]
```

You timed `soundex1c.py` already; this is how it performs:

```
C:\samples\soundex\stage1>python soundex1c.py
Woo             W000 14.5341678901
Pilgrim         P426 19.2650071448
Flingjingwaller F452 30.1003563302
```

This code is straightforward, but is it the best solution? Calling `upper()` on each individual character seems inefficient; it would probably be better to call `upper()` once on the entire string.

Then there's the matter of incrementally building the `digits` string. Incrementally building strings like this is horribly inefficient; internally, the Python interpreter needs to create a new string each time through the loop, then discard the old one.

Python is good at lists, though. It can treat a string as a list of characters automatically. And lists are easy to combine into strings again, using the string method `join()`.

Here is `soundex/stage2/soundex2a.py`, which converts letters to digits by using ¦ and `lambda`:

```
def soundex(source):
    # ...
    source = source.upper()
    digits = source[0] + "".join(map(lambda c: charToSoundex[c], source[1:]))
```

Surprisingly, `soundex2a.py` is not faster:

```
C:\samples\soundex\stage2>python soundex2a.py
Woo             W000 15.0097526362
Pilgrim         P426 19.254806407
Flingjingwaller F452 29.3790847719
```

The overhead of the anonymous `lambda` function kills any performance you gain by dealing with the string as a list of characters.

`soundex/stage2/soundex2b.py` uses a list comprehension instead of ¦ and `lambda`:

```
    source = source.upper()
    digits = source[0] + "".join([charToSoundex[c] for c in source[1:]])
```

Using a list comprehension in `soundex2b.py` is faster than using ¦ and `lambda` in `soundex2a.py`, but still not faster than the original code (incrementally building a string in `soundex1c.py`):

```
C:\samples\soundex\stage2>python soundex2b.py
Woo             W000 13.4221324219
Pilgrim         P426 16.4901234654
Flingjingwaller F452 25.8186157738
```

It's time for a radically different approach. Dictionary lookups are a general purpose tool. Dictionary keys can be any length string (or many other data types), but in this case we are only dealing with single−character keys *and* single−character values. It turns out that Python has a specialized function for handling exactly this situation: the `string.maketrans` function.

This is `soundex/stage2/soundex2c.py`:

```
allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
def soundex(source):
    # ...
    digits = source[0].upper() + source[1:].translate(charToSoundex)
```

What the heck is going on here? `string.maketrans` creates a translation matrix between two strings: the first argument and the second argument. In this case, the first argument is the string `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz`, and the second argument is the string `9123912992245591262391929291239129922455912623919292`. See the pattern? It's the same conversion pattern we were setting up longhand with a dictionary. A maps to 9, B maps to 1, C maps to 2, and so forth. But it's not a dictionary; it's a specialized data structure that you can access using the string method `translate`, which translates each character into the corresponding digit, according to the matrix defined by `string.maketrans`.

`timeit` shows that `soundex2c.py` is significantly faster than defining a dictionary and looping through the input and building the output incrementally:

```
C:\samples\soundex\stage2>python soundex2c.py
Woo             W000 11.437645008
Pilgrim         P426 13.2825062962
Flingjingwaller F452 18.5570110168
```

You're not going to get much better than that. Python has a specialized function that does exactly what you want to do; use it and move on.

Example 18.4. Best Result So Far: `soundex/stage2/soundex2c.py`

```
import string, re

allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
isOnlyChars = re.compile('^[A-Za-z]+$').search

def soundex(source):
    if not isOnlyChars(source):
        return "0000"
    digits = source[0].upper() + source[1:].translate(charToSoundex)
```

```
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

18.5. Optimizing List Operations

The third step in the Soundex algorithm is eliminating consecutive duplicate digits. What's the best way to do this?

Here's the code we have so far, in `soundex/stage2/soundex2c.py`:

```
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
```

Here are the performance results for `soundex2c.py`:

```
C:\samples\soundex\stage2>python soundex2c.py
Woo             W000 12.6070768771
Pilgrim         P426 14.4033353401
Flingjingwaller F452 19.7774882003
```

The first thing to consider is whether it's efficient to check `digits[-1]` each time through the loop. Are list indexes expensive? Would we be better off maintaining the last digit in a separate variable, and checking that instead?

To answer this question, here is `soundex/stage3/soundex3a.py`:

```
    digits2 = ''
    last_digit = ''
    for d in digits:
        if d != last_digit:
            digits2 += d
            last_digit = d
```

`soundex3a.py` does not run any faster than `soundex2c.py`, and may even be slightly slower (although it's not enough of a difference to say for sure):

```
C:\samples\soundex\stage3>python soundex3a.py
Woo             W000 11.5346048171
Pilgrim         P426 13.3950636184
Flingjingwaller F452 18.6108927252
```

Why isn't `soundex3a.py` faster? It turns out that list indexes in Python are extremely efficient. Repeatedly accessing `digits2[-1]` is no problem at all. On the other hand, manually maintaining the last seen digit in a separate variable means we have *two* variable assignments for each digit we're storing, which wipes out any small

gains we might have gotten from eliminating the list lookup.

Let's try something radically different. If it's possible to treat a string as a list of characters, it should be possible to use a list comprehension to iterate through the list. The problem is, the code needs access to the previous character in the list, and that's not easy to do with a straightforward list comprehension.

However, it is possible to create a list of index numbers using the built−in `range()` function, and use those index numbers to progressively search through the list and pull out each character that is different from the previous character. That will give you a list of characters, and you can use the string method `join()` to reconstruct a string from that.

Here is `soundex/stage3/soundex3b.py`:

```
    digits2 = "".join([digits[i] for i in range(len(digits))
                       if i == 0 or digits[i-1] != digits[i]])
```

Is this faster? In a word, no.

```
C:\samples\soundex\stage3>python soundex3b.py
Woo             W000 14.2245271396
Pilgrim         P426 17.8337165757
Flingjingwaller F452 25.9954005327
```

It's possible that the techniques so far as have been "string−centric". Python can convert a string into a list of characters with a single command: `list('abc')` returns `['a', 'b', 'c']`. Furthermore, lists can be *modified in place* very quickly. Instead of incrementally building a new list (or string) out of the source string, why not move elements around within a single list?

Here is `soundex/stage3/soundex3c.py`, which modifies a list in place to remove consecutive duplicate elements:

```
    digits = list(source[0].upper() + source[1:].translate(charToSoundex))
    i=0
    for item in digits:
        if item==digits[i]: continue
        i+=1
        digits[i]=item
    del digits[i+1:]
    digits2 = "".join(digits)
```

Is this faster than `soundex3a.py` or `soundex3b.py`? No, in fact it's the slowest method yet:

```
C:\samples\soundex\stage3>python soundex3c.py
Woo             W000 14.1662554878
Pilgrim         P426 16.0397885765
Flingjingwaller F452 22.1789341942
```

We haven't made any progress here at all, except to try and rule out several "clever" techniques. The fastest code we've seen so far was the original, most straightforward method (`soundex2c.py`). Sometimes it doesn't pay to be clever.

Example 18.5. Best Result So Far: `soundex/stage2/soundex2c.py`

```
import string, re
```

```
allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
isOnlyChars = re.compile('^[A-Za-z]+$').search

def soundex(source):
    if not isOnlyChars(source):
        return "0000"
    digits = source[0].upper() + source[1:].translate(charToSoundex)
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

18.6. Optimizing String Manipulation

The final step of the Soundex algorithm is padding short results with zeros, and truncating long results. What is the best way to do this?

This is what we have so far, taken from `soundex/stage2/soundex2c.py`:

```
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]
```

These are the results for `soundex2c.py`:

```
C:\samples\soundex\stage2>python soundex2c.py
Woo             W000 12.6070768771
Pilgrim         P426 14.4033353401
Flingjingwaller F452 19.7774882003
```

The first thing to consider is replacing that regular expression with a loop. This code is from `soundex/stage4/soundex4a.py`:

```
    digits3 = ''
    for d in digits2:
        if d != '9':
            digits3 += d
```

Is `soundex4a.py` faster? Yes it is:

```
C:\samples\soundex\stage4>python soundex4a.py
Woo             W000 6.62865531792
Pilgrim         P426 9.02247576158
Flingjingwaller F452 13.6328416042
```

But wait a minute. A loop to remove characters from a string? We can use a simple string method for that. Here's `soundex/stage4/soundex4b.py`:

```
    digits3 = digits2.replace('9', '')
```

Is `soundex4b.py` faster? That's an interesting question. It depends on the input:

```
C:\samples\soundex\stage4>python soundex4b.py
Woo             W000 6.75477414029
Pilgrim         P426 7.56652144337
Flingjingwaller F452 10.8727729362
```

The string method in `soundex4b.py` is faster than the loop for most names, but it's actually slightly slower than `soundex4a.py` in the trivial case (of a very short name). Performance optimizations aren't always uniform; tuning that makes one case faster can sometimes make other cases slower. In this case, the majority of cases will benefit from the change, so let's leave it at that, but the principle is an important one to remember.

Last but not least, let's examine the final two steps of the algorithm: padding short results with zeros, and truncating long results to four characters. The code you see in `soundex4b.py` does just that, but it's horribly inefficient. Take a look at `soundex/stage4/soundex4c.py` to see why:

```
    digits3 += '000'
    return digits3[:4]
```

Why do we need a `while` loop to pad out the result? We know in advance that we're going to truncate the result to four characters, and we know that we already have at least one character (the initial letter, which is passed unchanged from the original `source` variable). That means we can simply add three zeros to the output, then truncate it. Don't get stuck in a rut over the exact wording of the problem; looking at the problem slightly differently can lead to a simpler solution.

How much speed do we gain in `soundex4c.py` by dropping the `while` loop? It's significant:

```
C:\samples\soundex\stage4>python soundex4c.py
Woo             W000 4.89129791636
Pilgrim         P426 7.30642134685
Flingjingwaller F452 10.689832367
```

Finally, there is still one more thing you can do to these three lines of code to make them faster: you can combine them into one line. Take a look at `soundex/stage4/soundex4d.py`:

```
    return (digits2.replace('9', '') + '000')[:4]
```

Putting all this code on one line in `soundex4d.py` is barely faster than `soundex4c.py`:

```
C:\samples\soundex\stage4>python soundex4d.py
Woo             W000 4.93624105857
Pilgrim         P426 7.19747593619
Flingjingwaller F452 10.5490700634
```

It is also significantly less readable, and for not much performance gain. Is that worth it? I hope you have good comments. Performance isn't everything. Your optimization efforts must always be balanced against threats to your program's readability and maintainability.

18.7. Summary

This chapter has illustrated several important aspects of performance tuning in Python, and performance tuning in general.

- If you need to choose between regular expressions and writing a loop, choose regular expressions. The regular expression engine is compiled in C and runs natively on your computer; your loop is written in Python and runs through the Python interpreter.
- If you need to choose between regular expressions and string methods, choose string methods. Both are compiled in C, so choose the simpler one.
- General−purpose dictionary lookups are fast, but specialtiy functions such as `string.maketrans` and string methods such as `isalpha()` are faster. If Python has a custom−tailored function for you, use it.
- Don't be too clever. Sometimes the most obvious algorithm is also the fastest.
- Don't sweat it too much. Performance isn't everything.

I can't emphasize that last point strongly enough. Over the course of this chapter, you made this function three times faster and saved 20 seconds over 1 million function calls. Great. Now think: over the course of those million function calls, how many seconds will your surrounding application wait for a database connection? Or wait for disk I/O? Or wait for user input? Don't spend too much time over−optimizing one algorithm, or you'll ignore obvious improvements somewhere else. Develop an instinct for the sort of code that Python runs well, correct obvious blunders if you find them, and leave the rest alone.

Appendix A. Further reading

Chapter 1. Installing Python

Chapter 2. Your First Python Program

- 2.3. Documenting Functions
  - PEP 257 (http://www.python.org/peps/pep−0257.html) defines `doc string` conventions.
  - *Python Style Guide* (http://www.python.org/doc/essays/styleguide.html) discusses how to write a good `doc string`.
  - *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses conventions for spacing in `doc string`s (http://www.python.org/doc/current/tut/node6.html#SECTION006750000000000000000).
- 2.4.2. What's an Object?
  - *Python Reference Manual* (http://www.python.org/doc/current/ref/) explains exactly what it means to say that everything in Python is an object (http://www.python.org/doc/current/ref/objects.html), because some people are pedantic and like to discuss this sort of thing at great length.
  - eff−bot (http://www.effbot.org/guides/) summarizes Python objects (http://www.effbot.org/guides/python−objects.htm).
- 2.5. Indenting Code
  - *Python Reference Manual* (http://www.python.org/doc/current/ref/) discusses cross−platform indentation issues and shows various indentation errors (http://www.python.org/doc/current/ref/indentation.html).
  - *Python Style Guide* (http://www.python.org/doc/essays/styleguide.html) discusses good indentation style.
- 2.6. Testing Modules
  - *Python Reference Manual* (http://www.python.org/doc/current/ref/) discusses the low−level details of importing modules (http://www.python.org/doc/current/ref/import.html).

Chapter 3. Native Datatypes

- 3.1.3. Deleting Items From Dictionaries
  - *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) teaches about dictionaries and shows how to use dictionaries to model sparse matrices (http://www.ibiblio.org/obp/thinkCSpy/chap10.htm).
  - Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) has a lot of example code using dictionaries (http://www.faqts.com/knowledge−base/index.phtml/fid/541).
  - Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/) discusses how to sort the values of a dictionary by key (http://www.activestate.com/ASPN/Python/Cookbook/Recipe/52306).
  - *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes all the dictionary methods (http://www.python.org/doc/current/lib/typesmapping.html).
- 3.2.5. Using List Operators
  - *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) teaches about lists and makes an important point about passing lists as function arguments (http://www.ibiblio.org/obp/thinkCSpy/chap08.htm).

    - *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) shows how to use lists as stacks and queues (http://www.python.org/doc/current/tut/node7.html#SECTION007110000000000000000).
    - Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) answers common questions about lists (http://www.faqts.com/knowledge−base/index.phtml/fid/534) and has a lot of example code using lists (http://www.faqts.com/knowledge−base/index.phtml/fid/540).
    - *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes all the list methods (http://www.python.org/doc/current/lib/typesseq−mutable.html).
- 3.3. Introducing Tuples
    - *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) teaches about tuples and shows how to concatenate tuples (http://www.ibiblio.org/obp/thinkCSpy/chap10.htm).
    - Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) shows how to sort a tuple (http://www.faqts.com/knowledge−base/view.phtml/aid/4553/fid/587).
    - *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) shows how to define a tuple with one element (http://www.python.org/doc/current/tut/node7.html#SECTION007300000000000000000).
- 3.4.2. Assigning Multiple Values at Once
    - *Python Reference Manual* (http://www.python.org/doc/current/ref/) shows examples of when you can skip the line continuation character (http://www.python.org/doc/current/ref/implicit−joining.html) and when you need to use it (http://www.python.org/doc/current/ref/explicit−joining.html).
    - *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) shows how to use multi−variable assignment to swap the values of two variables (http://www.ibiblio.org/obp/thinkCSpy/chap09.htm).
- 3.5. Formatting Strings
    - *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes all the string formatting format characters (http://www.python.org/doc/current/lib/typesseq−strings.html).
    - *Effective AWK Programming* (http://www−gnats.gnu.org:8080/cgi−bin/info2www?(gawk)Top) discusses all the format characters (http://www−gnats.gnu.org:8080/cgi−bin/info2www?(gawk)Control+Letters) and advanced string formatting techniques like specifying width, precision, and zero−padding (http://www−gnats.gnu.org:8080/cgi−bin/info2www?(gawk)Format+Modifiers).
- 3.6. Mapping Lists
    - *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses another way to map lists using the built−in `map` function (http://www.python.org/doc/current/tut/node7.html#SECTION007130000000000000000).
    - *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) shows how to do nested list comprehensions (http://www.python.org/doc/current/tut/node7.html#SECTION007140000000000000000).
- 3.7. Joining Lists and Splitting Strings
    - Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) answers common questions about strings (http://www.faqts.com/knowledge−base/index.phtml/fid/480) and has a lot of example code using strings (http://www.faqts.com/knowledge−base/index.phtml/fid/539).
    - *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes all the string methods (http://www.python.org/doc/current/lib/string−methods.html).
    - *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the `string` module (http://www.python.org/doc/current/lib/module−string.html).
    - *The Whole Python FAQ* (http://www.python.org/doc/FAQ.html) explains why `join` is a string method

(http://www.python.org/cgi−bin/faqw.py?query=4.96&querytype=simple&casefold=yes&req=search) instead of a list method.

Chapter 4. The Power Of Introspection

- 4.2. Using Optional and Named Arguments
    - *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses exactly when and how default arguments are evaluated (http://www.python.org/doc/current/tut/node6.html#SECTION006710000000000000000), which matters when the default value is a list or an expression with side effects.
- 4.3.3. Built−In Functions
    - *Python Library Reference* (http://www.python.org/doc/current/lib/) documents all the built−in functions (http://www.python.org/doc/current/lib/built−in−funcs.html) and all the built−in exceptions (http://www.python.org/doc/current/lib/module−exceptions.html).
- 4.5. Filtering Lists
    - *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses another way to filter lists using the built−in `filter` function (http://www.python.org/doc/current/tut/node7.html#SECTION007130000000000000000).
- 4.6.1. Using the and−or Trick
    - Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/) discusses alternatives to the `and−or` trick (http://www.activestate.com/ASPN/Python/Cookbook/Recipe/52310).
- 4.7.1. Real−World lambda Functions
    - Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) discusses using `lambda` to call functions indirectly (http://www.faqts.com/knowledge−base/view.phtml/aid/6081/fid/241).
    - *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) shows how to access outside variables from inside a `lambda` function (http://www.python.org/doc/current/tut/node6.html#SECTION006740000000000000000). (PEP 227 (http://python.sourceforge.net/peps/pep−0227.html) explains how this will change in future versions of Python.)
    - *The Whole Python FAQ* (http://www.python.org/doc/FAQ.html) has examples of obfuscated one−liners using `lambda` (http://www.python.org/cgi−bin/faqw.py?query=4.15&querytype=simple&casefold=yes&req=search).

Chapter 5. Objects and Object−Orientation

- 5.2. Importing Modules Using from module import
    - eff−bot (http://www.effbot.org/guides/) has more to say on `import` *`module`* *vs.* `from` *`module`* `import` (http://www.effbot.org/guides/import−confusion.htm).
    - *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses advanced import techniques, including `from` *`module`* `import *` (http://www.python.org/doc/current/tut/node8.html#SECTION008410000000000000000).
- 5.3.2. Knowing When to Use self and __init__
    - *Learning to Program* (http://www.freenetpages.co.uk/hp/alan.gauld/) has a gentler introduction to classes (http://www.freenetpages.co.uk/hp/alan.gauld/tutclass.htm).

* ♦ *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) shows how to use classes to model compound datatypes (http://www.ibiblio.org/obp/thinkCSpy/chap12.htm).
  ♦ *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) has an in−depth look at classes, namespaces, and inheritance (http://www.python.org/doc/current/tut/node11.html).
  ♦ Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) answers common questions about classes (http://www.faqts.com/knowledge−base/index.phtml/fid/242).
* 5.4.1. Garbage Collection

  ♦ *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes built−in attributes like `__class__` (http://www.python.org/doc/current/lib/specialattrs.html).
  ♦ *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the `gc` module (http://www.python.org/doc/current/lib/module−gc.html), which gives you low−level control over Python's garbage collection.
* 5.5. Exploring UserDict: A Wrapper Class

  ♦ *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the `UserDict` module (http://www.python.org/doc/current/lib/module−UserDict.html) and the `copy` module (http://www.python.org/doc/current/lib/module−copy.html).
* 5.7. Advanced Special Class Methods

  ♦ *Python Reference Manual* (http://www.python.org/doc/current/ref/) documents all the special class methods (http://www.python.org/doc/current/ref/specialnames.html).
* 5.9. Private Functions

  ♦ *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses the inner workings of private variables (http://www.python.org/doc/current/tut/node11.html#SECTION0011600000000000000000).

Chapter 6. Exceptions and File Handling

* 6.1.1. Using Exceptions For Other Purposes

  ♦ *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses defining and raising your own exceptions, and handling multiple exceptions at once (http://www.python.org/doc/current/tut/node10.html#SECTION0010400000000000000000).
  ♦ *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes all the built−in exceptions (http://www.python.org/doc/current/lib/module−exceptions.html).
  ♦ *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the getpass (http://www.python.org/doc/current/lib/module−getpass.html) module.
  ♦ *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the `traceback` module (http://www.python.org/doc/current/lib/module−traceback.html), which provides low−level access to exception attributes after an exception is raised.
  ♦ *Python Reference Manual* (http://www.python.org/doc/current/ref/) discusses the inner workings of the `try...except` block (http://www.python.org/doc/current/ref/try.html).
* 6.2.4. Writing to Files

  ♦ *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses reading and writing files, including how to read a file one line at a time into a list (http://www.python.org/doc/current/tut/node9.html#SECTION0092100000000000000000).
  ♦ eff−bot (http://www.effbot.org/guides/) discusses efficiency and performance of various ways of reading a file (http://www.effbot.org/guides/readline−performance.htm).
  ♦ Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) answers

common questions about files (http://www.faqts.com/knowledge−base/index.phtml/fid/552).
    - *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes all the file object methods (http://www.python.org/doc/current/lib/bltin−file−objects.html).
- 6.4. Using sys.modules
    - *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) discusses exactly when and how default arguments are evaluated (http://www.python.org/doc/current/tut/node6.html#SECTION006710000000000000000).
    - *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the `sys` (http://www.python.org/doc/current/lib/module−sys.html) module.
- 6.5. Working with Directories
    - Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) answers questions about the `os` module (http://www.faqts.com/knowledge−base/index.phtml/fid/240).
    - *Python Library Reference* (http://www.python.org/doc/current/lib/) documents the `os` (http://www.python.org/doc/current/lib/module−os.html) module and the `os.path` (http://www.python.org/doc/current/lib/module−os.path.html) module.

Chapter 7. Regular Expressions

- 7.6. Case study: Parsing Phone Numbers
    - Regular Expression HOWTO (http://py−howto.sourceforge.net/regex/regex.html) teaches about regular expressions and how to use them in Python.
    - *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes the `re` module (http://www.python.org/doc/current/lib/module−re.html).

Chapter 8. HTML Processing

- 8.4. Introducing BaseHTMLProcessor.py
    - W3C (http://www.w3.org/) discusses character and entity references (http://www.w3.org/TR/REC−html40/charset.html#entities).
    - *Python Library Reference* (http://www.python.org/doc/current/lib/) confirms your suspicions that the `htmlentitydefs` module (http://www.python.org/doc/current/lib/module−htmlentitydefs.html) is exactly what it sounds like.
- 8.9. Putting it all together
    - You thought I was kidding about the server−side scripting idea. So did I, until I found this web−based dialectizer (http://rinkworks.com/dialect/). Unfortunately, source code does not appear to be available.

Chapter 9. XML Processing

- 9.4. Unicode
    - Unicode.org (http://www.unicode.org/) is the home page of the unicode standard, including a brief technical introduction (http://www.unicode.org/standard/principles.html).
    - Unicode Tutorial (http://www.reportlab.com/i18n/python_unicode_tutorial.html) has some more examples of how to use Python's unicode functions, including how to force Python to coerce unicode into ASCII even when it doesn't really want to.
    - PEP 263 (http://www.python.org/peps/pep−0263.html) goes into more detail about how and when to define a character encoding in your `.py` files.

Chapter 10. Scripts and Streams

Chapter 11. HTTP Web Services

- 11.1. Diving in
    - Paul Prescod believes that pure HTTP web services are the future of the Internet (http://webservices.xml.com/pub/a/ws/2002/02/06/rest.html).

Chapter 12. SOAP Web Services

- 12.1. Diving In
    - http://www.xmethods.net/ is a repository of public access SOAP web services.
    - The SOAP specification (http://www.w3.org/TR/soap/) is surprisingly readable, if you like that sort of thing.
- 12.8. Troubleshooting SOAP Web Services
    - New developments for SOAPpy (http://www−106.ibm.com/developerworks/webservices/library/ws−pyth17.html) steps through trying to connect to another SOAP service that doesn't quite work as advertised.

Chapter 13. Unit Testing

- 13.1. Introduction to Roman numerals
    - This site (http://www.wilkiecollins.demon.co.uk/roman/front.htm) has more on Roman numerals, including a fascinating history (http://www.wilkiecollins.demon.co.uk/roman/intro.htm) of how Romans and other civilizations really used them (short answer: haphazardly and inconsistently).
- 13.3. Introducing romantest.py
    - The PyUnit home page (http://pyunit.sourceforge.net/) has an in−depth discussion of using the `unittest` framework (http://pyunit.sourceforge.net/pyunit.html), including advanced features not covered in this chapter.
    - The PyUnit FAQ (http://pyunit.sourceforge.net/pyunit.html) explains why test cases are stored separately (http://pyunit.sourceforge.net/pyunit.html#WHERE) from the code they test.
    - *Python Library Reference* (http://www.python.org/doc/current/lib/) summarizes the `unittest` (http://www.python.org/doc/current/lib/module−unittest.html) module.
    - ExtremeProgramming.org (http://www.extremeprogramming.org/) discusses why you should write unit tests (http://www.extremeprogramming.org/rules/unittests.html).
    - The Portland Pattern Repository (http://www.c2.com/cgi/wiki) has an ongoing discussion of unit tests (http://www.c2.com/cgi/wiki?UnitTests), including a standard definition (http://www.c2.com/cgi/wiki?StandardDefinitionOfUnitTest), why you should code unit tests first (http://www.c2.com/cgi/wiki?CodeUnitTestFirst), and several in−depth case studies (http://www.c2.com/cgi/wiki?UnitTestTrial).

Chapter 14. Test−First Programming

Chapter 15. Refactoring

- 15.5. Summary

        ♦ XProgramming.com (http://www.xprogramming.com/) has links to download unit testing frameworks (http://www.xprogramming.com/software.htm) for many different languages.

Chapter 16. Functional Programming

Chapter 17. Dynamic functions

- 17.7. plural.py, stage 6

    - PEP 255 (http://www.python.org/peps/pep−0255.html) defines generators.
    - Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/) has many more examples of generators (http://www.google.com/search?q=generators+cookbook+site:aspn.activestate.com).

Chapter 18. Performance Tuning

- 18.1. Diving in

    - Soundexing and Genealogy (http://www.avotaynu.com/soundex.html) gives a chronology of the evolution of the Soundex and its regional variations.

Appendix B. A 5−minute review

Chapter 1. Installing Python

- 1.1. Which Python is right for you?

  The first thing you need to do with Python is install it. Or do you?
- 1.2. Python on Windows

  On Windows, you have a couple choices for installing Python.
- 1.3. Python on Mac OS X

  On Mac OS X, you have two choices for installing Python: install it, or don't install it. You probably want to install it.
- 1.4. Python on Mac OS 9

  Mac OS 9 does not come with any version of Python, but installation is very simple, and there is only one choice.
- 1.5. Python on RedHat Linux

  Download the latest Python RPM by going to http://www.python.org/ftp/python/ and selecting the highest version number listed, then selecting the `rpms/` directory within that. Then download the RPM with the highest version number. You can install it with the **rpm** command, as shown here:
- 1.6. Python on Debian GNU/Linux

  If you are lucky enough to be running Debian GNU/Linux, you install Python through the **apt** command.
- 1.7. Python Installation from Source

  If you prefer to build from source, you can download the Python source code from http://www.python.org/ftp/python/. Select the highest version number listed, download the `.tgz` file), and then do the usual **`configure`**, **`make`**, **`make install`** dance.
- 1.8. The Interactive Shell

  Now that you have Python installed, what's this interactive shell thing you're running?
- 1.9. Summary

  You should now have a version of Python installed that works for you.

Chapter 2. Your First Python Program

- 2.1. Diving in

  Here is a complete, working Python program.
- 2.2. Declaring Functions

  Python has functions like most other languages, but it does not have separate header files like C++ or `interface`/`implementation` sections like Pascal. When you need a function, just declare it, like this:
- 2.3. Documenting Functions

You can document a Python function by giving it a `doc string`.

* 2.4. Everything Is an Object

  A function, like everything else in Python, is an object.
* 2.5. Indenting Code

  Python functions have no explicit `begin` or `end`, and no curly braces to mark where the function code starts and stops. The only delimiter is a colon (`:`) and the indentation of the code itself.
* 2.6. Testing Modules

  Python modules are objects and have several useful attributes. You can use this to easily test your modules as you write them. Here's an example that uses the `if __name__` trick.

Chapter 3. Native Datatypes

* 3.1. Introducing Dictionaries

  One of Python's built−in datatypes is the dictionary, which defines one−to−one relationships between keys and values.
* 3.2. Introducing Lists

  Lists are Python's workhorse datatype. If your only experience with lists is arrays in Visual Basic or (God forbid) the datastore in Powerbuilder, brace yourself for Python lists.
* 3.3. Introducing Tuples

  A tuple is an immutable list. A tuple can not be changed in any way once it is created.
* 3.4. Declaring variables

  Python has local and global variables like most other languages, but it has no explicit variable declarations. Variables spring into existence by being assigned a value, and they are automatically destroyed when they go out of scope.
* 3.5. Formatting Strings

  Python supports formatting values into strings. Although this can include very complicated expressions, the most basic usage is to insert values into a string with the `%s` placeholder.
* 3.6. Mapping Lists

  One of the most powerful features of Python is the list comprehension, which provides a compact way of mapping a list into another list by applying a function to each of the elements of the list.
* 3.7. Joining Lists and Splitting Strings

  You have a list of key−value pairs in the form *`key=value`*, and you want to join them into a single string. To join any list of strings into a single string, use the `join` method of a string object.
* 3.8. Summary

  The `odbchelper.py` program and its output should now make perfect sense.

Chapter 4. The Power Of Introspection

* 4.1. Diving In

  Here is a complete, working Python program. You should understand a good deal about it just by looking at it. The numbered lines illustrate concepts covered in Chapter 2, *Your First Python Program*. Don't worry if the rest of the code looks intimidating; you'll learn all about it throughout this chapter.
* 4.2. Using Optional and Named Arguments

  Python allows function arguments to have default values; if the function is called without the argument, the argument gets its default value. Futhermore, arguments can be specified in any order by using named arguments. Stored procedures in SQL Server Transact/SQL can do this, so if you're a SQL Server scripting guru, you can skim this part.
* 4.3. Using type, str, dir, and Other Built−In Functions

  Python has a small set of extremely useful built−in functions. All other functions are partitioned off into modules. This was actually a conscious design decision, to keep the core language from getting bloated like other scripting languages (cough cough, Visual Basic).
* 4.4. Getting Object References With getattr

  You already know that Python functions are objects. What you don't know is that you can get a reference to a function without knowing its name until run−time, by using the `getattr` function.
* 4.5. Filtering Lists

  As you know, Python has powerful capabilities for mapping lists into other lists, via list comprehensions (Section 3.6, Mapping Lists). This can be combined with a filtering mechanism, where some elements in the list are mapped while others are skipped entirely.
* 4.6. The Peculiar Nature of and and or

  In Python, `and` and `or` perform boolean logic as you would expect, but they do not return boolean values; instead, they return one of the actual values they are comparing.
* 4.7. Using lambda Functions

  Python supports an interesting syntax that lets you define one−line mini−functions on the fly. Borrowed from Lisp, these so−called `lambda` functions can be used anywhere a function is required.
* 4.8. Putting It All Together

  The last line of code, the only one you haven't deconstructed yet, is the one that does all the work. But by now the work is easy, because everything you need is already set up just the way you need it. All the dominoes are in place; it's time to knock them down.
* 4.9. Summary

  The `apihelper.py` program and its output should now make perfect sense.

Chapter 5. Objects and Object−Orientation

* 5.1. Diving In

  Here is a complete, working Python program. Read the `doc string`s of the module, the classes, and the functions to get an overview of what this program does and how it works. As usual, don't worry about the stuff you don't understand; that's what the rest of the chapter is

for.

- 5.2. Importing Modules Using from module import

  Python has two ways of importing modules. Both are useful, and you should know when to use each. One way, `import module`, you've already seen in Section 2.4, Everything Is an Object . The other way accomplishes the same thing, but it has subtle and important differences.
- 5.3. Defining Classes

  Python is fully object−oriented: you can define your own classes, inherit from your own or built−in classes, and instantiate the classes you've defined.
- 5.4. Instantiating Classes

  Instantiating classes in Python is straightforward. To instantiate a class, simply call the class as if it were a function, passing the arguments that the `__init__` method defines. The return value will be the newly created object.
- 5.5. Exploring UserDict: A Wrapper Class

  As you've seen, `FileInfo` is a class that acts like a dictionary. To explore this further, let's look at the `UserDict` class in the `UserDict` module, which is the ancestor of the `FileInfo` class. This is nothing special; the class is written in Python and stored in a `.py` file, just like any other Python code. In particular, it's stored in the `lib` directory in your Python installation.
- 5.6. Special Class Methods

  In addition to normal class methods, there are a number of special methods that Python classes can define. Instead of being called directly by your code (like normal methods), special methods are called for you by Python in particular circumstances or when specific syntax is used.
- 5.7. Advanced Special Class Methods

  Python has more special methods than just `__getitem__` and `__setitem__`. Some of them let you emulate functionality that you may not even know about.
- 5.8. Introducing Class Attributes

  You already know about data attributes, which are variables owned by a specific instance of a class. Python also supports class attributes, which are variables owned by the class itself.
- 5.9. Private Functions

  Unlike in most languages, whether a Python function, method, or attribute is private or public is determined entirely by its name.
- 5.10. Summary

  That's it for the hard−core object trickery. You'll see a real−world application of special class methods in Chapter 12, which uses `getattr` to create a proxy to a remote web service.

Chapter 6. Exceptions and File Handling

- 6.1. Handling Exceptions

  Like many other programming languages, Python has exception handling via `try...except` blocks.

* 6.2. Working with File Objects

  Python has a built−in function, `open`, for opening a file on disk. `open` returns a file object, which has methods and attributes for getting information about and manipulating the opened file.
* 6.3. Iterating with for Loops

  Like most other languages, Python has `for` loops. The only reason you haven't seen them until now is that Python is good at so many other things that you don't need them as often.
* 6.4. Using sys.modules

  Modules, like everything else in Python, are objects. Once imported, you can always get a reference to a module through the global dictionary `sys.modules`.
* 6.5. Working with Directories

  The `os.path` module has several functions for manipulating files and directories. Here, we're looking at handling pathnames and listing the contents of a directory.
* 6.6. Putting It All Together

  Once again, all the dominoes are in place. You've seen how each line of code works. Now let's step back and see how it all fits together.
* 6.7. Summary

  The `fileinfo.py` program introduced in Chapter 5 should now make perfect sense.

Chapter 7. Regular Expressions

* 7.1. Diving In

  If what you're trying to do can be accomplished with string functions, you should use them. They're fast and simple and easy to read, and there's a lot to be said for fast, simple, readable code. But if you find yourself using a lot of different string functions with `if` statements to handle special cases, or if you're combining them with `split` and `join` and list comprehensions in weird unreadable ways, you may need to move up to regular expressions.
* 7.2. Case Study: Street Addresses

  This series of examples was inspired by a real−life problem I had in my day job several years ago, when I needed to scrub and standardize street addresses exported from a legacy system before importing them into a newer system. (See, I don't just make this stuff up; it's actually useful.) This example shows how I approached the problem.
* 7.3. Case Study: Roman Numerals

  You've most likely seen Roman numerals, even if you didn't recognize them. You may have seen them in copyrights of old movies and television shows ("Copyright `MCMXLVI`" instead of "Copyright `1946`"), or on the dedication walls of libraries or universities ("established `MDCCCLXXXVIII`" instead of "established `1888`"). You may also have seen them in outlines and bibliographical references. It's a system of representing numbers that really does date back to the ancient Roman empire (hence the name).
* 7.4. Using the {n,m} Syntax

  In the previous section, you were dealing with a pattern where the same character could be repeated up to three times. There is another way to express this in regular expressions, which

some people find more readable. First look at the method we already used in the previous example.

- 7.5. Verbose Regular Expressions

  So far you've just been dealing with what I'll call "compact" regular expressions. As you've seen, they are difficult to read, and even if you figure out what one does, that's no guarantee that you'll be able to understand it six months later. What you really need is inline documentation.
- 7.6. Case study: Parsing Phone Numbers

  So far you've concentrated on matching whole patterns. Either the pattern matches, or it doesn't. But regular expressions are much more powerful than that. When a regular expression *does* match, you can pick out specific pieces of it. You can find out what matched where.
- 7.7. Summary

  This is just the tiniest tip of the iceberg of what regular expressions can do. In other words, even though you're completely overwhelmed by them now, believe me, you ain't seen nothing yet.

Chapter 8. HTML Processing

- 8.1. Diving in

  I often see questions on comp.lang.python (http://groups.google.com/groups?group=comp.lang.python) like "How can I list all the [headers|images|links] in my HTML document?" "How do I parse/translate/munge the text of my HTML document but leave the tags alone?" "How can I add/remove/quote attributes of all my HTML tags at once?" This chapter will answer all of these questions.
- 8.2. Introducing sgmllib.py

  HTML processing is broken into three steps: breaking down the HTML into its constituent pieces, fiddling with the pieces, and reconstructing the pieces into HTML again. The first step is done by `sgmllib.py`, a part of the standard Python library.
- 8.3. Extracting data from HTML documents

  To extract data from HTML documents, subclass the `SGMLParser` class and define methods for each tag or entity you want to capture.
- 8.4. Introducing BaseHTMLProcessor.py

  `SGMLParser` doesn't produce anything by itself. It parses and parses and parses, and it calls a method for each interesting thing it finds, but the methods don't do anything. `SGMLParser` is an HTML *consumer*: it takes HTML and breaks it down into small, structured pieces. As you saw in the previous section, you can subclass `SGMLParser` to define classes that catch specific tags and produce useful things, like a list of all the links on a web page. Now you'll take this one step further by defining a class that catches everything `SGMLParser` throws at it and reconstructs the complete HTML document. In technical terms, this class will be an HTML *producer*.
- 8.5. locals and globals

  Let's digress from HTML processing for a minute and talk about how Python handles variables. Python has two built−in functions, `locals` and `globals`, which provide

dictionary−based access to local and global variables.

- 8.6. Dictionary−based string formatting

  There is an alternative form of string formatting that uses dictionaries instead of tuples of values.
- 8.7. Quoting attribute values

  A common question on comp.lang.python (http://groups.google.com/groups?group=comp.lang.python) is "I have a bunch of HTML documents with unquoted attribute values, and I want to properly quote them all. How can I do this?"[4] (This is generally precipitated by a project manager who has found the HTML−is−a−standard religion joining a large project and proclaiming that all pages must validate against an HTML validator. Unquoted attribute values are a common violation of the HTML standard.) Whatever the reason, unquoted attribute values are easy to fix by feeding HTML through `BaseHTMLProcessor`.
- 8.8. Introducing dialect.py

  `Dialectizer` is a simple (and silly) descendant of `BaseHTMLProcessor`. It runs blocks of text through a series of substitutions, but it makes sure that anything within a `<pre>...</pre>` block passes through unaltered.
- 8.9. Putting it all together

  It's time to put everything you've learned so far to good use. I hope you were paying attention.
- 8.10. Summary

  Python provides you with a powerful tool, `sgmllib.py`, to manipulate HTML by turning its structure into an object model. You can use this tool in many different ways.

Chapter 9. XML Processing

- 9.1. Diving in

  There are two basic ways to work with XML. One is called SAX ("Simple API for XML"), and it works by reading the XML a little bit at a time and calling a method for each element it finds. (If you read Chapter 8, *HTML Processing*, this should sound familiar, because that's how the `sgmllib` module works.) The other is called DOM ("Document Object Model"), and it works by reading in the entire XML document at once and creating an internal representation of it using native Python classes linked in a tree structure. Python has standard modules for both kinds of parsing, but this chapter will only deal with using the DOM.
- 9.2. Packages

  Actually parsing an XML document is very simple: one line of code. However, before you get to that line of code, you need to take a short detour to talk about packages.
- 9.3. Parsing XML

  As I was saying, actually parsing an XML document is very simple: one line of code. Where you go from there is up to you.
- 9.4. Unicode

  Unicode is a system to represent characters from all the world's different languages. When Python parses an XML document, all data is stored in memory as unicode.
- 9.5. Searching for elements

Traversing XML documents by stepping through each node can be tedious. If you're looking for something in particular, buried deep within your XML document, there is a shortcut you can use to find it quickly: `getElementsByTagName`.

* 9.6. Accessing element attributes

  XML elements can have one or more attributes, and it is incredibly simple to access them once you have parsed an XML document.
* 9.7. Segue

  OK, that's it for the hard−core XML stuff. The next chapter will continue to use these same example programs, but focus on other aspects that make the program more flexible: using streams for input processing, using `getattr` for method dispatching, and using command−line flags to allow users to reconfigure the program without changing the code.

Chapter 10. Scripts and Streams

* 10.1. Abstracting input sources

  One of Python's greatest strengths is its dynamic binding, and one powerful use of dynamic binding is the *file−like object*.
* 10.2. Standard input, output, and error

  UNIX users are already familiar with the concept of standard input, standard output, and standard error. This section is for the rest of you.
* 10.3. Caching node lookups

  `kgp.py` employs several tricks which may or may not be useful to you in your XML processing. The first one takes advantage of the consistent structure of the input documents to build a cache of nodes.
* 10.4. Finding direct children of a node

  Another useful techique when parsing XML documents is finding all the direct child elements of a particular element. For instance, in the grammar files, a `ref` element can have several `p` elements, each of which can contain many things, including other `p` elements. You want to find just the `p` elements that are children of the `ref`, not `p` elements that are children of other `p` elements.
* 10.5. Creating separate handlers by node type

  The third useful XML processing tip involves separating your code into logical functions, based on node types and element names. Parsed XML documents are made up of various types of nodes, each represented by a Python object. The root level of the document itself is represented by a `Document` object. The `Document` then contains one or more `Element` objects (for actual XML tags), each of which may contain other `Element` objects, `Text` objects (for bits of text), or `Comment` objects (for embedded comments). Python makes it easy to write a dispatcher to separate the logic for each node type.
* 10.6. Handling command−line arguments

  Python fully supports creating programs that can be run on the command line, complete with command−line arguments and either short− or long−style flags to specify various options. None of this is XML−specific, but this script makes good use of command−line processing, so it seemed like a good time to mention it.
* 10.7. Putting it all together

You've covered a lot of ground. Let's step back and see how all the pieces fit together.

- 10.8. Summary

  Python comes with powerful libraries for parsing and manipulating XML documents. The `minidom` takes an XML file and parses it into Python objects, providing for random access to arbitrary elements. Furthermore, this chapter shows how Python can be used to create a "real" standalone command−line script, complete with command−line flags, command−line arguments, error handling, even the ability to take input from the piped result of a previous program.

Chapter 11. HTTP Web Services

- 11.1. Diving in

  You've learned about HTML processing and XML processing, and along the way you saw how to download a web page and how to parse XML from a URL, but let's dive into the more general topic of HTTP web services.
- 11.2. How not to fetch data over HTTP

  Let's say you want to download a resource over HTTP, such as a syndicated Atom feed. But you don't just want to download it once; you want to download it over and over again, every hour, to get the latest news from the site that's offering the news feed. Let's do it the quick−and−dirty way first, and then see how you can do better.
- 11.3. Features of HTTP

  There are five important features of HTTP which you should support.
- 11.4. Debugging HTTP web services

  First, let's turn on the debugging features of Python's HTTP library and see what's being sent over the wire. This will be useful throughout the chapter, as you add more and more features.
- 11.5. Setting the User−Agent

  The first step to improving your HTTP web services client is to identify yourself properly with a `User−Agent`. To do that, you need to move beyond the basic `urllib` and dive into `urllib2`.
- 11.6. Handling Last−Modified and ETag

  Now that you know how to add custom HTTP headers to your web service requests, let's look at adding support for `Last−Modified` and `ETag` headers.
- 11.7. Handling redirects

  You can support permanent and temporary redirects using a different kind of custom URL handler.
- 11.8. Handling compressed data

  The last important HTTP feature you want to support is compression. Many web services have the ability to send data compressed, which can cut down the amount of data sent over the wire by 60% or more. This is especially true of XML web services, since XML data compresses very well.
- 11.9. Putting it all together

  You've seen all the pieces for building an intelligent HTTP web services client. Now let's see

In previous chapters, you "dived in" by immediately looking at code and trying to understand it as quickly as possible. Now that you have some Python under your belt, you're going to step back and look at the steps that happen *before* the code gets written.

- 13.2. Diving in

  Now that you've completely defined the behavior you expect from your conversion functions, you're going to do something a little unexpected: you're going to write a test suite that puts these functions through their paces and makes sure that they behave the way you want them to. You read that right: you're going to write code that tests code that you haven't written yet.
- 13.3. Introducing romantest.py

  This is the complete test suite for your Roman numeral conversion functions, which are yet to be written but will eventually be in `roman.py`. It is not immediately obvious how it all fits together; none of these classes or methods reference any of the others. There are good reasons for this, as you'll see shortly.
- 13.4. Testing for success

  The most fundamental part of unit testing is constructing individual test cases. A test case answers a single question about the code it is testing.
- 13.5. Testing for failure

  It is not enough to test that functions succeed when given good input; you must also test that they fail when given bad input. And not just any sort of failure; they must fail in the way you expect.
- 13.6. Testing for sanity

  Often, you will find that a unit of code contains a set of reciprocal functions, usually in the form of conversion functions where one converts A to B and the other converts B to A. In these cases, it is useful to create a "sanity check" to make sure that you can convert A to B and back to A without losing precision, incurring rounding errors, or triggering any other sort of bug.

Chapter 14. Test−First Programming

- 14.1. roman.py, stage 1

  Now that the unit tests are complete, it's time to start writing the code that the test cases are attempting to test. You're going to do this in stages, so you can see all the unit tests fail, then watch them pass one by one as you fill in the gaps in `roman.py`.
- 14.2. roman.py, stage 2

  Now that you have the framework of the `roman` module laid out, it's time to start writing code and passing test cases.
- 14.3. roman.py, stage 3

  Now that `toRoman` behaves correctly with good input (integers from `1` to `3999`), it's time to make it behave correctly with bad input (everything else).
- 14.4. roman.py, stage 4

  Now that `toRoman` is done, it's time to start coding `fromRoman`. Thanks to the rich data structure that maps individual Roman numerals to integer values, this is no more difficult than the `toRoman` function.

* 14.5. roman.py, stage 5

  Now that `fromRoman` works properly with good input, it's time to fit in the last piece of the puzzle: making it work properly with bad input. That means finding a way to look at a string and determine if it's a valid Roman numeral. This is inherently more difficult than validating numeric input in `toRoman`, but you have a powerful tool at your disposal: regular expressions.

Chapter 15. Refactoring

* 15.1. Handling bugs

  Despite your best efforts to write comprehensive unit tests, bugs happen. What do I mean by "bug"? A bug is a test case you haven't written yet.
* 15.2. Handling changing requirements

  Despite your best efforts to pin your customers to the ground and extract exact requirements from them on pain of horrible nasty things involving scissors and hot wax, requirements will change. Most customers don't know what they want until they see it, and even if they do, they aren't that good at articulating what they want precisely enough to be useful. And even if they do, they'll want more in the next release anyway. So be prepared to update your test cases as requirements change.
* 15.3. Refactoring

  The best thing about comprehensive unit testing is not the feeling you get when all your test cases finally pass, or even the feeling you get when someone else blames you for breaking their code and you can actually *prove* that you didn't. The best thing about unit testing is that it gives you the freedom to refactor mercilessly.
* 15.4. Postscript

  A clever reader read the previous section and took it to the next level. The biggest headache (and performance drain) in the program as it is currently written is the regular expression, which is required because you have no other way of breaking down a Roman numeral. But there's only 5000 of them; why don't you just build a lookup table once, then simply read that? This idea gets even better when you realize that you don't need to use regular expressions at all. As you build the lookup table for converting integers to Roman numerals, you can build the reverse lookup table to convert Roman numerals to integers.
* 15.5. Summary

  Unit testing is a powerful concept which, if properly implemented, can both reduce maintenance costs and increase flexibility in any long−term project. It is also important to understand that unit testing is not a panacea, a Magic Problem Solver, or a silver bullet. Writing good test cases is hard, and keeping them up to date takes discipline (especially when customers are screaming for critical bug fixes). Unit testing is not a replacement for other forms of testing, including functional testing, integration testing, and user acceptance testing. But it is feasible, and it does work, and once you've seen it work, you'll wonder how you ever got along without it.

Chapter 16. Functional Programming

* 16.1. Diving in

In Chapter 13, *Unit Testing*, you learned about the philosophy of unit testing. In Chapter 14, *Test−First Programming*, you stepped through the implementation of basic unit tests in Python. In Chapter 15, *Refactoring*, you saw how unit testing makes large−scale refactoring easier. This chapter will build on those sample programs, but here we will focus more on advanced Python−specific techniques, rather than on unit testing itself.

* 16.2. Finding the path

  When running Python scripts from the command line, it is sometimes useful to know where the currently running script is located on disk.
* 16.3. Filtering lists revisited

  You're already familiar with using list comprehensions to filter lists. There is another way to accomplish this same thing, which some people feel is more expressive.
* 16.4. Mapping lists revisited

  You're already familiar with using list comprehensions to map one list into another. There is another way to accomplish the same thing, using the built−in `map` function. It works much the same way as the `filter` function.
* 16.5. Data−centric programming

  By now you're probably scratching your head wondering why this is better than using `for` loops and straight function calls. And that's a perfectly valid question. Mostly, it's a matter of perspective. Using `map` and `filter` forces you to center your thinking around your data.
* 16.6. Dynamically importing modules

  OK, enough philosophizing. Let's talk about dynamically importing modules.
* 16.7. Putting it all together

  You've learned enough now to deconstruct the first seven lines of this chapter's code sample: reading a directory and importing selected modules within it.
* 16.8. Summary

  The `regression.py` program and its output should now make perfect sense.

Chapter 17. Dynamic functions

* 17.1. Diving in

  I want to talk about plural nouns. Also, functions that return other functions, advanced regular expressions, and generators. Generators are new in Python 2.3. But first, let's talk about how to make plural nouns.
* 17.2. plural.py, stage 1

  So you're looking at words, which at least in English are strings of characters. And you have rules that say you need to find different combinations of characters, and then do different things to them. This sounds like a job for regular expressions.
* 17.3. plural.py, stage 2

  Now you're going to add a level of abstraction. You started by defining a list of rules: if this, then do that, otherwise go to the next rule. Let's temporarily complicate part of the program so you can simplify another part.
* 17.4. plural.py, stage 3

Appendix C. Tips and tricks

Chapter 1. Installing Python

Chapter 2. Your First Python Program

- 2.1. Diving in

  In the ActivePython IDE on Windows, you can run the Python program you're editing by choosing File−>Run... (**Ctrl−R**). Output is displayed in the interactive window.

  In the Python IDE on Mac OS, you can run a Python program with Python−>Run window... (**Cmd−R**), but there is an important option you must set first. Open the `.py` file in the IDE, pop up the options menu by clicking the black triangle in the upper−right corner of the window, and make sure the Run as __main__ option is checked. This is a per−file setting, but you'll only need to do it once per file.

  On UNIX−compatible systems (including Mac OS X), you can run a Python program from the command line: **`python odbchelper.py`**
- 2.2. Declaring Functions

  In Visual Basic, functions (that return a value) start with `function`, and subroutines (that do not return a value) start with `sub`. There are no subroutines in Python. Everything is a function, all functions return a value (even if it's `None`), and all functions start with `def`.

  In Java, C++, and other statically−typed languages, you must specify the datatype of the function return value and each function argument. In Python, you never explicitly specify the datatype of anything. Based on what value you assign, Python keeps track of the datatype internally.
- 2.3. Documenting Functions

  Triple quotes are also an easy way to define a string with both single and double quotes, like `qq/.../` in Perl.

  Many Python IDEs use the `doc string` to provide context−sensitive documentation, so that when you type a function name, its `doc string` appears as a tooltip. This can be incredibly helpful, but it's only as good as the `doc string`s you write.
- 2.4. Everything Is an Object

  `import` in Python is like `require` in Perl. Once you `import` a Python module, you access its functions with *`module.function`*; once you `require` a Perl module, you access its functions with *`module::function`*.
- 2.5. Indenting Code

  Python uses carriage returns to separate statements and a colon and indentation to separate code blocks. C++ and Java use semicolons to separate statements and curly braces to separate code blocks.
- 2.6. Testing Modules

  Like C, Python uses == for comparison and = for assignment. Unlike C, Python does not support in−line assignment, so there's no chance of accidentally assigning the value you thought you were comparing.

  On MacPython, there is an additional step to make the `if __name__` trick work. Pop up the module's options menu by clicking the black triangle in the upper−right corner of the window, and make sure Run as __main__ is checked.

Chapter 3. Native Datatypes

- 3.1. Introducing Dictionaries

A dictionary in Python is like a hash in Perl. In Perl, variables that store hashes always start with a `%` character. In Python, variables can be named anything, and Python keeps track of the datatype internally.

A dictionary in Python is like an instance of the `Hashtable` class in Java.

A dictionary in Python is like an instance of the `Scripting.Dictionary` object in Visual Basic.

- 3.1.2. Modifying Dictionaries

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered. This is an important distinction that will annoy you when you want to access the elements of a dictionary in a specific, repeatable order (like alphabetical order by key). There are ways of doing this, but they're not built into the dictionary.

- 3.2. Introducing Lists

A list in Python is like an array in Perl. In Perl, variables that store arrays always start with the `@` character; in Python, variables can be named anything, and Python keeps track of the datatype internally.

A list in Python is much more than an array in Java (although it can be used as one if that's really all you want out of life). A better analogy would be to the `ArrayList` class, which can hold arbitrary objects and can expand dynamically as new items are added.

- 3.2.3. Searching Lists

Before version 2.2.1, Python had no separate boolean datatype. To compensate for this, Python accepted almost anything in a boolean context (like an `if` statement), according to the following rules:

  - `0` is false; all other numbers are true.
  - An empty string (`""`) is false, all other strings are true.
  - An empty list (`[]`) is false; all other lists are true.
  - An empty tuple (`()`) is false; all other tuples are true.
  - An empty dictionary (`{}`) is false; all other dictionaries are true.

These rules still apply in Python 2.2.1 and beyond, but now you can also use an actual boolean, which has a value of `True` or `False`. Note the capitalization; these values, like everything else in Python, are case−sensitive.

- 3.3. Introducing Tuples

Tuples can be converted into lists, and vice−versa. The built−in `tuple` function takes a list and returns a tuple with the same elements, and the `list` function takes a tuple and returns a list. In effect, `tuple` freezes a list, and `list` thaws a tuple.

- 3.4. Declaring variables

When a command is split among several lines with the line−continuation marker ("\"), the continued lines can be indented in any manner; Python's normally stringent indentation rules do not apply. If your Python IDE auto−indents the continued line, you should probably accept its default unless you have a burning reason not to.

- 3.5. Formatting Strings

String formatting in Python uses the same syntax as the `sprintf` function in C.

- 3.7. Joining Lists and Splitting Strings

`join` works only on lists of strings; it does not do any type coercion. Joining a list that has one or more non−string elements will raise an exception.

`anystring.split(delimiter, 1)` is a useful technique when you want to search a string for a substring and then work with everything before the substring (which ends up in the first element of the returned list) and everything after it (which ends up in the second element).

Chapter 4. The Power Of Introspection

- 4.2. Using Optional and Named Arguments

The only thing you need to do to call a function is specify a value (somehow) for each required argument; the manner and order in which you do that is up to you.

* 4.3.3. Built−In Functions

  Python comes with excellent reference manuals, which you should peruse thoroughly to learn all the modules Python has to offer. But unlike most languages, where you would find yourself referring back to the manuals or man pages to remind yourself how to use these modules, Python is largely self−documenting.
* 4.7. Using lambda Functions

  `lambda` functions are a matter of style. Using them is never required; anywhere you could use them, you could define a separate normal function and use that instead. I use them in places where I want to encapsulate specific, non−reusable code without littering my code with a lot of little one−line functions.
* 4.8. Putting It All Together

  In SQL, you must use `IS NULL` instead of `= NULL` to compare a null value. In Python, you can use either `== None` or `is None`, but `is None` is faster.

Chapter 5. Objects and Object−Orientation

* 5.2. Importing Modules Using from module import

  `from` *`module`* `import *` in Python is like `use` *`module`* in Perl; `import` *`module`* in Python is like `require` *`module`* in Perl.

  `from` *`module`* `import *` in Python is like `import` *`module`*`.*` in Java; `import` *`module`* in Python is like `import` *`module`* in Java.

  Use `from module import *` sparingly, because it makes it difficult to determine where a particular function or attribute came from, and that makes debugging and refactoring more difficult.
* 5.3. Defining Classes

  The `pass` statement in Python is like an empty set of braces (`{}`) in Java or C.

  In Python, the ancestor of a class is simply listed in parentheses immediately after the class name. There is no special keyword like `extends` in Java.
* 5.3.1. Initializing and Coding Classes

  By convention, the first argument of any Python class method (the reference to the current instance) is called `self`. This argument fills the role of the reserved word `this` in C++ or Java, but `self` is not a reserved word in Python, merely a naming convention. Nonetheless, please don't call it anything but `self`; this is a very strong convention.
* 5.3.2. Knowing When to Use self and __init__

  `__init__` methods are optional, but when you define one, you must remember to explicitly call the ancestor's `__init__` method (if it defines one). This is more generally true: whenever a descendant wants to extend the behavior of the ancestor, the descendant method must explicitly call the ancestor method at the proper time, with the proper arguments.
* 5.4. Instantiating Classes

  In Python, simply call a class as if it were a function to create a new instance of the class. There is no explicit `new` operator like C++ or Java.
* 5.5. Exploring UserDict: A Wrapper Class

  In the ActivePython IDE on Windows, you can quickly open any module in your library path by selecting File−>Locate... (**Ctrl−L**).

  Java and Powerbuilder support function overloading by argument list, *i.e.* one class can have multiple methods with the same name but a different number of arguments, or arguments of different types. Other languages (most notably PL/SQL) even support function overloading by argument name; *i.e.* one class can

have multiple methods with the same name and the same number of arguments of the same type but different argument names. Python supports neither of these; it has no form of function overloading whatsoever. Methods are defined solely by their name, and there can be only one method per class with a given name. So if a descendant class has an `__init__` method, it *always* overrides the ancestor `__init__` method, even if the descendant defines it with a different argument list. And the same rule applies to any other method.

Guido, the original author of Python, explains method overriding this way: "Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class, may in fact end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)" If that doesn't make sense to you (it confuses the hell out of me), feel free to ignore it. I just thought I'd pass it along.

Always assign an initial value to all of an instance's data attributes in the `__init__` method. It will save you hours of debugging later, tracking down `AttributeError` exceptions because you're referencing uninitialized (and therefore non−existent) attributes.

In versions of Python prior to 2.2, you could not directly subclass built−in datatypes like strings, lists, and dictionaries. To compensate for this, Python comes with wrapper classes that mimic the behavior of these built−in datatypes: `UserString`, `UserList`, and `UserDict`. Using a combination of normal and special methods, the `UserDict` class does an excellent imitation of a dictionary. In Python 2.2 and later, you can inherit classes directly from built−in datatypes like `dict`. An example of this is given in the examples that come with this book, in `fileinfo_fromdict.py`.

- 5.6.1. Getting and Setting Items

  When accessing data attributes within a class, you need to qualify the attribute name: `self.`*`attribute`*. When calling other methods within a class, you need to qualify the method name: `self.`*`method`*.

- 5.7. Advanced Special Class Methods

  In Java, you determine whether two string variables reference the same physical memory location by using `str1 == str2`. This is called *object identity*, and it is written in Python as `str1 is str2`. To compare string values in Java, you would use `str1.equals(str2)`; in Python, you would use `str1 == str2`. Java programmers who have been taught to believe that the world is a better place because == in Java compares by identity instead of by value may have a difficult time adjusting to Python's lack of such "gotchas".

  While other object−oriented languages only let you define the physical model of an object ("this object has a `GetLength` method"), Python's special class methods like `__len__` allow you to define the logical model of an object ("this object has a length").

- 5.8. Introducing Class Attributes

  In Java, both static variables (called class attributes in Python) and instance variables (called data attributes in Python) are defined immediately after the class definition (one with the `static` keyword, one without). In Python, only class attributes can be defined here; data attributes are defined in the `__init__` method.

  There are no constants in Python. Everything can be changed if you try hard enough. This fits with one of the core principles of Python: bad behavior should be discouraged but not banned. If you really want to change the value of `None`, you can do it, but don't come running to me when your code is impossible to debug.

- 5.9. Private Functions

  In Python, all special methods (like `__setitem__`) and built−in attributes (like `__doc__`) follow a standard naming convention: they both start with and end with two underscores. Don't name your own methods and attributes this way, because it will only confuse you (and others) later.

Chapter 6. Exceptions and File Handling

- 6.1. Handling Exceptions

Python uses `try...except` to handle exceptions and `raise` to generate them. Java and C++ use `try...catch` to handle exceptions, and `throw` to generate them.

- 6.5. Working with Directories

  Whenever possible, you should use the functions in `os` and `os.path` for file, directory, and path manipulations. These modules are wrappers for platform−specific modules, so functions like `os.path.split` work on UNIX, Windows, Mac OS, and any other platform supported by Python.

Chapter 7. Regular Expressions

- 7.4. Using the {n,m} Syntax

  There is no way to programmatically determine that two regular expressions are equivalent. The best you can do is write a lot of test cases to make sure they behave the same way on all relevant inputs. You'll talk more about writing test cases later in this book.

Chapter 8. HTML Processing

- 8.2. Introducing sgmllib.py

  Python 2.0 had a bug where `SGMLParser` would not recognize declarations at all (`handle_decl` would never be called), which meant that `DOCTYPE`s were silently ignored. This is fixed in Python 2.1.

  In the ActivePython IDE on Windows, you can specify command line arguments in the "Run script" dialog. Separate multiple arguments with spaces.
- 8.4. Introducing BaseHTMLProcessor.py

  The HTML specification requires that all non−HTML (like client−side JavaScript) must be enclosed in HTML comments, but not all web pages do this properly (and all modern web browsers are forgiving if they don't). `BaseHTMLProcessor` is not forgiving; if script is improperly embedded, it will be parsed as if it were HTML. For instance, if the script contains less−than and equals signs, `SGMLParser` may incorrectly think that it has found tags and attributes. `SGMLParser` always converts tags and attribute names to lowercase, which may break the script, and `BaseHTMLProcessor` always encloses attribute values in double quotes (even if the original HTML document used single quotes or no quotes), which will certainly break the script. Always protect your client−side script within HTML comments.
- 8.5. locals and globals

  Python 2.2 introduced a subtle but important change that affects the namespace search order: nested scopes. In versions of Python prior to 2.2, when you reference a variable within a nested function or `lambda` function, Python will search for that variable in the current (nested or `lambda`) function's namespace, then in the module's namespace. Python 2.2 will search for the variable in the current (nested or `lambda`) function's namespace, *then in the parent function's namespace*, then in the module's namespace. Python 2.1 can work either way; by default, it works like Python 2.0, but you can add the following line of code at the top of your module to make your module work like Python 2.2:

  ```
  from __future__ import nested_scopes
  ```

  Using the `locals` and `globals` functions, you can get the value of arbitrary variables dynamically, providing the variable name as a string. This mirrors the functionality of the `getattr` function, which allows you to access arbitrary functions dynamically by providing the function name as a string.
- 8.6. Dictionary−based string formatting

  Using dictionary−based string formatting with `locals` is a convenient way of making complex string formatting expressions more readable, but it comes with a price. There is a slight performance hit in making the call to `locals`, since `locals` builds a copy of the local namespace.

Chapter 9. XML Processing

* 9.2. Packages

  A package is a directory with the special `__init__.py` file in it. The `__init__.py` file defines the attributes and methods of the package. It doesn't need to define anything; it can just be an empty file, but it has to exist. But if `__init__.py` doesn't exist, the directory is just a directory, not a package, and it can't be imported or contain modules or nested packages.
* 9.6. Accessing element attributes

  This section may be a little confusing, because of some overlapping terminology. Elements in an XML document have attributes, and Python objects also have attributes. When you parse an XML document, you get a bunch of Python objects that represent all the pieces of the XML document, and some of these Python objects represent attributes of the XML elements. But the (Python) objects that represent the (XML) attributes also have (Python) attributes, which are used to access various parts of the (XML) attribute that the object represents. I told you it was confusing. I am open to suggestions on how to distinguish these more clearly.

  Like a dictionary, attributes of an XML element have no ordering. Attributes may *happen to be* listed in a certain order in the original XML document, and the `Attr` objects may *happen to be* listed in a certain order when the XML document is parsed into Python objects, but these orders are arbitrary and should carry no special meaning. You should always access individual attributes by name, like the keys of a dictionary.

Chapter 10. Scripts and Streams

Chapter 11. HTTP Web Services

* 11.6. Handling Last−Modified and ETag

  In these examples, the HTTP server has supported both `Last-Modified` and `ETag` headers, but not all servers do. As a web services client, you should be prepared to support both, but you must code defensively in case a server only supports one or the other, or neither.

Chapter 12. SOAP Web Services

Chapter 13. Unit Testing

* 13.2. Diving in

  `unittest` is included with Python 2.1 and later. Python 2.0 users can download it from `pyunit.sourceforge.net` (http://pyunit.sourceforge.net/).

Chapter 14. Test−First Programming

* 14.3. roman.py, stage 3

  The most important thing that comprehensive unit testing can tell you is when to stop coding. When all the unit tests for a function pass, stop coding the function. When all the unit tests for an entire module pass, stop coding the module.
* 14.5. roman.py, stage 5

  When all of your tests pass, stop coding.

Chapter 15. Refactoring

* 15.3. Refactoring

Whenever you are going to use a regular expression more than once, you should compile it to get a pattern object, then call the methods on the pattern object directly.

Chapter 16. Functional Programming

* 16.2. Finding the path

  The pathnames and filenames you pass to `os.path.abspath` do not need to exist.

  `os.path.abspath` not only constructs full path names, it also normalizes them. That means that if you are in the `/usr/` directory, `os.path.abspath('bin/../local/bin')` will return `/usr/local/bin`. It normalizes the path by making it as simple as possible. If you just want to normalize a pathname like this without turning it into a full pathname, use `os.path.normpath` instead.

  Like the other functions in the `os` and `os.path` modules, `os.path.abspath` is cross−platform. Your results will look slightly different than my examples if you're running on Windows (which uses backslash as a path separator) or Mac OS (which uses colons), but they'll still work. That's the whole point of the `os` module.

Chapter 17. Dynamic functions

Chapter 18. Performance Tuning

* 18.2. Using the timeit Module

  You can use the `timeit` module on the command line to test an existing Python program, without modifying the code. See http://docs.python.org/lib/node396.html for documentation on the command−line flags.

  The `timeit` module only works if you already know what piece of code you need to optimize. If you have a larger Python program and don't know where your performance problems are, check out the `hotshot` module. (http://docs.python.org/lib/module−hotshot.html)

Appendix D. List of examples

Chapter 5. Objects and Object−Orientation

Chapter 6. Exceptions and File Handling

Appendix E. Revision history

Revision History

Revision 5.4 — 2004−05−20

- Added Section 12.1, Diving In .
- Added Section 12.2, Installing the SOAP Libraries .
- Added Section 12.3, First Steps with SOAP .
- Added Section 12.4, Debugging SOAP Web Services .
- Added Section 12.5, Introducing WSDL .
- Added Section 12.6, Introspecting SOAP Web Services with WSDL .
- Added Section 12.7, Searching Google .
- Added Section 12.8, Troubleshooting SOAP Web Services .
- Added Section 12.9, Summary .
- Incorporated technical reviewer revisions in Chapter 16, *Functional Programming* and Chapter 18, *Performance Tuning*.

Revision 5.3 — 2004−05−12

- Added `isalpha()` example to Section 18.3, Optimizing Regular Expressions . Thanks, Paul.
- Incorporated copyediting revisions into Chapter 5, *Objects and Object−Orientation* and Chapter 6, *Exceptions and File Handling*.
- Fixed URL of Section 9.7, Segue .

Revision 5.2 — 2004−05−09

- Fixed URL of Section 14.1, roman.py, stage 1 .
- Added Section 18.1, Diving in .
- Added Section 18.2, Using the timeit Module .
- Added Section 18.3, Optimizing Regular Expressions .
- Added Section 18.4, Optimizing Dictionary Lookups .
- Added Section 18.5, Optimizing List Operations .
- Added Section 18.6, Optimizing String Manipulation .
- Added Section 18.7, Summary .

Revision 5.1 — 2004−05−05

- Clarified Example 7.7, Checking for Tens and Example 7.8, Validating Roman Numerals with {n,m} .
- Clarified Example 7.10, Finding Numbers .
- Fixed typo in Example 11.6, Testing Last−Modified . Thanks, Jesir.
- Fixed typo in Example 3.11, The Difference between extend and append . Thanks, Daniel.
- Incorporated technical reviewer revisions.
- Incorporated copy editor revisions in Chapter 1, *Installing Python*, Chapter 2, *Your First Python Program*, Chapter 3, *Native Datatypes*, and Chapter 4, *The Power Of Introspection*.

Revision 5.0 — 2004−04−16

- Added Section 11.1, Diving in .
- Added Section 11.2, How not to fetch data over HTTP .
- Added Section 11.3, Features of HTTP .
- Added Section 11.4, Debugging HTTP web services .
- Added Section 11.5, Setting the User−Agent .
- Added Section 11.6, Handling Last−Modified and ETag .

* Added Section 11.7, Handling redirects .
* Added Section 11.8, Handling compressed data .
* Added Section 11.9, Putting it all together .
* Added Section 11.10, Summary .
* Added Example 3.11, The Difference between extend and append .
* Changed descriptions of how to download Python throughout Chapter 1, *Installing Python* to be more generic and less version−specific.
* Changed references of "module" to "program" in Section 2.1, Diving in and Section 2.4, Everything Is an Object since we haven't explained modules yet.
* Added explicit instructions in Section 2.4, Everything Is an Object for the reader to open their Python IDE and follow along with the examples.
* Changed all examples and descriptions that referred to truth values `1` and `0` to refer to `True` and `False`.
* Updated Example 3.22, String Formatting vs. Concatenating to show new Python 2.3 `TypeError` message.
* Fixed typo in Example 17.19, Using generators instead of recursion .
* Fixed typo in Section 7.7, Summary .
* Fixed typo in Example 17.9, plural4.py .

Revision 4.9	2004−03−25

* Finished Section 16.7, Putting it all together .
* Added Section 16.8, Summary .
* Split unit testing introduction into two chapters, Chapter 13, *Unit Testing* and Chapter 14, *Test−First Programming*.
* Fixed typo in Example 17.12, plural4.py, finishing up .
* Fixed typo in Example 17.18, Introducing generators .

Revision 4.8	2004−03−25

* Finished Section 17.7, plural.py, stage 6 .
* Finished Section 17.8, Summary .
* Fixed broken links in Appendix A, *Further reading*, Appendix B, *A 5−minute review*, Appendix C, *Tips and tricks*, Appendix D, *List of examples*.

Revision 4.7	2004−03−21

* Added Section 17.1, Diving in .
* Added Section 17.2, plural.py, stage 1 .
* Added Section 17.3, plural.py, stage 2 .
* Added Section 17.4, plural.py, stage 3 .
* Added Section 17.5, plural.py, stage 4 .
* Added Section 17.6, plural.py, stage 5 .
* Added Section 17.7, plural.py, stage 6 (unfinished).
* Added Section 17.8, Summary (unfinished).

Revision 4.6	2004−03−14

* Finished Section 7.4, Using the {n,m} Syntax .
* Finished Section 7.5, Verbose Regular Expressions .
* Finished Section 7.6, Case study: Parsing Phone Numbers .
* Expanded Section 7.7, Summary .

Revision 4.5	2004−03−07

* Added Section 7.1, Diving In .

- Added Section 7.4, Using the {n,m} Syntax (incomplete).
- Added Section 7.5, Verbose Regular Expressions (incomplete).
- Added Section 7.6, Case study: Parsing Phone Numbers (incomplete).
- Added Section 7.7, Summary .
- Moved Section 7.2, Case Study: Street Addresses and Section 7.3, Case Study: Roman Numerals to regular expressions chapter.
- Added Example 6.20, Listing Directories with glob .
- Added Example 6.7, Writing to Files .
- Added Example 5.11, Inheriting Directly from Built−In Datatype dict .
- Added Example 10.11, Printing to stderr .
- Added Example 4.12, Creating a Dispatcher with getattr and Example 4.13, getattr Default Values .
- Added Example 2.6, if Statements .
- Added Example 3.23, Formatting Numbers .
- Split Chapter 5, *Objects and Object−Orientation* into 2 chapters: Chapter 5, *Objects and Object−Orientation* and Chapter 6, *Exceptions and File Handling*.
- Split Chapter 9, *XML Processing* into 2 chapters: Chapter 9, *XML Processing* and Chapter 10, *Scripts and Streams*.
- Split Chapter 13, *Unit Testing* into 2 chapters: Chapter 13, *Unit Testing* and Chapter 15, *Refactoring*.
- Renamed `help` to `info` in Chapter 4, *The Power Of Introspection*.
- Fixed incorrect back−reference in Section 8.5, locals and globals .
- Fixed broken example links in Section 8.1, Diving in .
- Fixed missing line in example in Section 9.1, Diving in .
- Fixed typo in Section 8.2, Introducing sgmllib.py .

Revision 4.4	2003−10−08

- Added Section 1.1, Which Python is right for you? .
- Added Section 1.2, Python on Windows .
- Added Section 1.3, Python on Mac OS X .
- Added Section 1.4, Python on Mac OS 9 .
- Added Section 1.5, Python on RedHat Linux .
- Added Section 1.6, Python on Debian GNU/Linux .
- Added Section 1.7, Python Installation from Source .
- Added Section 1.9, Summary .
- Removed preface.
- Fixed typo in Example 3.27, Output of odbchelper.py .
- Added link to PEP 257 in Section 2.3, Documenting Functions .
- Fixed link to *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) in Section 3.4.2, Assigning Multiple Values at Once .
- Added note about implied assert in Section 3.3, Introducing Tuples .

Revision 4.3	2003−09−28

- Added Section 16.6, Dynamically importing modules .
- Added Section 16.7, Putting it all together (incomplete).
- Fixed links in Appendix F, *About the book*.

Revision 4.2.1	2003−09−17

- Fixed links on index page.
- Fixed syntax highlighting.

Revision 4.2	2003−09−12

* Fixed typos in Section 16.4, Mapping lists revisited , Section 16.3, Filtering lists revisited , Section 7.2, Case Study: Street Addresses , and Section 10.6, Handling command−line arguments . Thanks, Doug.
* Fixed external link in Section 5.3, Defining Classes . Thanks, Harry.
* Changed wording at the end of Section 4.5, Filtering Lists . Thanks, Paul.
* Added sentence in Section 13.5, Testing for failure to make it clearer that we're passing a function to `assertRaises`, not a function name as a string. Thanks, Stephen.
* Fixed typo in Section 8.8, Introducing dialect.py . Thanks, Wellie.
* Fixed links to dialectized examples.
* Fixed external link to the history of Roman numerals. Thanks to many concerned Roman numeral fans around the world.

Revision 4.1	2002−07−28

* Added Section 10.3, Caching node lookups .
* Added Section 10.4, Finding direct children of a node .
* Added Section 10.5, Creating separate handlers by node type .
* Added Section 10.6, Handling command−line arguments .
* Added Section 10.7, Putting it all together .
* Added Section 10.8, Summary .
* Fixed typo in Section 6.5, Working with Directories . It's `os.getcwd()`, not `os.path.getcwd()`. Thanks, Abhishek.
* Fixed typo in Section 3.7, Joining Lists and Splitting Strings . When evaluated (instead of printed), the Python IDE will display single quotes around the output.
* Changed `str` example in Section 4.8, Putting It All Together to use a user−defined function, since Python 2.2 obsoleted the old example by defining a `doc string` for the built−in dictionary methods. Thanks Eric.
* Fixed typo in Section 9.4, Unicode , "anyway" to "anywhere". Thanks Frank.
* Fixed typo in Section 13.6, Testing for sanity , doubled word "accept". Thanks Ralph.
* Fixed typo in Section 15.3, Refactoring , `C?C?C?` matches 0 to 3 `C` characters, not 4. Thanks Ralph.
* Clarified and expanded explanation of implied slice indices in Example 3.9, Slicing Shorthand . Thanks Petr.
* Added historical note in Section 5.5, Exploring UserDict: A Wrapper Class now that Python 2.2 supports subclassing built−in datatypes directly.
* Added explanation of `update` dictionary method in Example 5.9, Defining the UserDict Class . Thanks Petr.
* Clarified Python's lack of overloading in Section 5.5, Exploring UserDict: A Wrapper Class . Thanks Petr.
* Fixed typo in Example 8.8, Introducing BaseHTMLProcessor . HTML comments end with two dashes and a bracket, not one. Thanks Petr.
* Changed tense of note about nested scopes in Section 8.5, locals and globals now that Python 2.2 is out. Thanks Petr.
* Fixed typo in Example 8.14, Dictionary−based string formatting in BaseHTMLProcessor.py ; a space should have been a non−breaking space. Thanks Petr.
* Added title to note on derived classes in Section 5.5, Exploring UserDict: A Wrapper Class . Thanks Petr.
* Added title to note on downloading `unittest` in Section 15.3, Refactoring . Thanks Petr.
* Fixed typesetting problem in Example 16.6, Running scripts in the current directory ; tabs should have been spaces, and the line numbers were misaligned. Thanks Petr.
* Fixed capitalization typo in the tip on truth values in Section 3.2, Introducing Lists . It's `True` and `False`, not `true` and `false`. Thanks to everyone who pointed this out.
* Changed section titles of Section 3.1, Introducing Dictionaries , Section 3.2, Introducing Lists , and Section 3.3, Introducing Tuples . "Dictionaries 101" was a cute way of saying that this section was an beginner's introduction to dictionaries. American colleges tend to use this numbering scheme to indicate introductory courses with no prerequisites, but apparently this is a distinctly American tradition, and it was unnecessarily confusing my international readers. In my defense, when I initially wrote these sections a year and a half ago, it never occurred to me that I would have international readers.

- Upgraded to version 1.52 of the DocBook XSL stylesheets.
- Upgraded to version 6.52 of the SAXON XSLT processor from Michael Kay.
- Various accessibility−related stylesheet tweaks.
- Somewhere between this revision and the last one, she said yes. The wedding will be next spring.

Revision 4.0−2	2002−04−26

- Fixed typo in Example 4.15, Introducing and .
- Fixed typo in Example 2.4, Import Search Path .
- Fixed Windows help file (missing table of contents due to base stylesheet changes).

Revision 4.0	2002−04−19

- Expanded Section 2.4, Everything Is an Object to include more about import search paths.
- Fixed typo in Example 3.7, Negative List Indices . Thanks to Brian for the correction.
- Rewrote the tip on truth values in Section 3.2, Introducing Lists , now that Python has a separate boolean datatype.
- Fixed typo in Section 5.2, Importing Modules Using from module import when comparing syntax to Java. Thanks to Rick for the correction.
- Added note in Section 5.5, Exploring UserDict: A Wrapper Class about derived classes always overriding ancestor classes.
- Fixed typo in Example 5.18, Modifying Class Attributes . Thanks to Kevin for the correction.
- Added note in Section 6.1, Handling Exceptions that you can define and raise your own exceptions. Thanks to Rony for the suggestion.
- Fixed typo in Example 8.17, Handling specific tags . Thanks for Rick for the correction.
- Added note in Example 8.18, SGMLParser about what the return codes mean. Thanks to Howard for the suggestion.
- Added `str` function when creating `StringIO` instance in Example 10.6, openAnything . Thanks to Ganesan for the idea.
- Added link in Section 13.3, Introducing romantest.py to explanation of why test cases belong in a separate file.
- Changed Section 16.2, Finding the path to use `os.path.dirname` instead of `os.path.split`. Thanks to Marc for the idea.
- Added code samples (`piglatin.py`, `parsephone.py`, and `plural.py`) for the upcoming regular expressions chapter.
- Updated and expanded list of Python distributions on home page.

Revision 3.9	2002−01−01

- Added Section 9.4, Unicode .
- Added Section 9.5, Searching for elements .
- Added Section 9.6, Accessing element attributes .
- Added Section 10.1, Abstracting input sources .
- Added Section 10.2, Standard input, output, and error .
- Added simple counter `for` loop examples (good usage and bad usage) in Section 6.3, Iterating with for Loops . Thanks to Kevin for the idea.
- Fixed typo in Example 3.25, The keys, values, and items Functions (two elements of `params.values()` were reversed).
- Fixed mistake in Section 4.3, Using type, str, dir, and Other Built−In Functions with regards to the name of the `__builtin__` module. Thanks to Denis for the correction.
- Added additional example in Section 16.2, Finding the path to show how to run unit tests in the current working directory, instead of the directory where `regression.py` is located.
- Modified explanation of how to derive a negative list index from a positive list index in Example 3.7, Negative List Indices . Thanks to Renauld for the suggestion.

- Updated links on home page for downloading latest version of Python. - Added link on home page to Bruce Eckel's preliminary draft of *Thinking in Python* (http://www.mindview.net/Books/TIPython), a marvelous (and advanced) book on design patterns and how to implement them in Python.	
Revision 3.8	2001−11−18
- Added Section 16.2, Finding the path . - Added Section 16.3, Filtering lists revisited . - Added Section 16.4, Mapping lists revisited . - Added Section 16.5, Data−centric programming . - Expanded sample output in Section 16.1, Diving in . - Finished Section 9.3, Parsing XML .	
Revision 3.7	2001−09−30
- Added Section 9.2, Packages . - Added Section 9.3, Parsing XML . - Cleaned up introductory paragraph in Section 9.1, Diving in . Thanks to Matt for this suggestion. - Added Java tip in Section 5.2, Importing Modules Using from module import . Thanks to Ori for this suggestion. - Fixed mistake in Section 4.8, Putting It All Together where I implied that you could not use `is None` to compare to a null value in Python. In fact, you can, and it's faster than `== None`. Thanks to Ori pointing this out. - Clarified in Section 3.2, Introducing Lists where I said that `li = li + other` was equivalent to `li.extend(other)`. The result is the same, but `extend` is faster because it doesn't create a new list. Thanks to Denis pointing this out. - Fixed mistake in Section 3.2, Introducing Lists where I said that `li += other` was equivalent to `li = li + other`. In fact, it's equivalent to `li.extend(other)`, since it doesn't create a new list. Thanks to Denis pointing this out. - Fixed typographical laziness in Chapter 2, *Your First Python Program*; when I was writing it, I had not yet standardized on putting string literals in single quotes within the text. They were set off by typography, but this is lost in some renditions of the book (like plain text), making it difficult to read. Thanks to Denis for this suggestion. - Fixed mistake in Section 2.2, Declaring Functions where I said that statically typed languages always use explicit variable + datatype declarations to enforce static typing. Most do, but there are some statically typed languages where the compiler figures out what type the variable is based on usage within the code. Thanks to Tony for pointing this out. - Added link to Spanish translation (http://es.diveintopython.org/).	
Revision 3.6.4	2001−09−06
- Added code in `BaseHTMLProcessor` to handle non−HTML entity references, and added a note about it in Section 8.4, Introducing BaseHTMLProcessor.py . - Modified Example 8.11, Introducing globals to include `htmlentitydefs` in the output.	
Revision 3.6.3	2001−09−04
- Fixed typo in Section 9.1, Diving in . - Added link to Korean translation (http://kr.diveintopython.org/html/index.htm).	
Revision 3.6.2	2001−08−31
Fixed typo in Section 13.6, Testing for sanity (the last requirement was listed twice).	
Revision 3.6	2001−08−31

- Finished Chapter 8, *HTML Processing* with Section 8.9, Putting it all together and Section 8.10, Summary .
- Added Section 15.4, Postscript .
- Started Chapter 9, *XML Processing* with Section 9.1, Diving in .
- Started Chapter 16, *Functional Programming* with Section 16.1, Diving in .
- Fixed long−standing bug in colorizing script that improperly colorized the examples in Chapter 8, *HTML Processing*.
- Added link to French translation (http://fr.diveintopython.org/toc.html). They did the right thing and translated the source XML, so they can re−use all my build scripts and make their work available in six different formats.
- Upgraded to version 1.43 of the DocBook XSL stylesheets.
- Upgraded to version 6.43 of the SAXON XSLT processor from Michael Kay.
- Massive stylesheet changes, moving away from a table−based layout and towards more appropriate use of cascading style sheets. Unfortunately, CSS has as many compatibility problems as anything else, so there are still some tables used in the header and footer. The resulting HTML version looks worse in Netscape 4, but better in modern browsers, including Netscape 6, Mozilla, Internet Explorer 5, Opera 5, Konqueror, and iCab. And it's still completely readable in Lynx. I love Lynx. It was my first web browser. You never forget your first.
- Moved to Ant (http://jakarta.apache.org/ant/) to have better control over the build process, which is especially important now that I'm juggling six output formats and two languages.
- Consolidated the available downloadable archives; previously, I had different files for each platform, because the .zip files that Python's `zipfile` module creates are non−standard and can't be opened by Aladdin Expander on Mac OS. But the .zip files that Ant creates are completely standard and cross−platform. Go Ant!
- Now hosting the complete XML source, XSL stylesheets, and associated scripts and libraries on SourceForge. There's also CVS access for the really adventurous.
- Re−licensed the example code under the new−and−improved GPL−compatible Python 2.1.1 license (http://www.python.org/2.1.1/license.html). Thanks, Guido; people really do care, and it really does matter.

Revision 3.5	2001−06−26

- Added explanation of strong/weak/static/dynamic datatypes in Section 2.2, Declaring Functions .
- Added case−sensitivity example in Section 3.1, Introducing Dictionaries .
- Use `os.path.normcase` in Chapter 5, *Objects and Object−Orientation* to compensate for inferior operating systems whose files aren't case−sensitive.
- Fixed indentation problems in code samples in PDF version.

Revision 3.4	2001−05−31

- Added Section 14.5, roman.py, stage 5 .
- Added Section 15.1, Handling bugs .
- Added Section 15.2, Handling changing requirements .
- Added Section 15.3, Refactoring .
- Added Section 15.5, Summary .
- Fixed yet another stylesheet bug that was dropping nested `</span>` tags.

Revision 3.3	2001−05−24

- Added Section 13.2, Diving in .
- Added Section 13.3, Introducing romantest.py .
- Added Section 13.4, Testing for success .
- Added Section 13.5, Testing for failure .
- Added Section 13.6, Testing for sanity .
- Added Section 14.1, roman.py, stage 1 .
- Added Section 14.2, roman.py, stage 2 .

- Added Section 14.3, roman.py, stage 3 .
- Added Section 14.4, roman.py, stage 4 .
- Tweaked stylesheets in an endless quest for complete Netscape/Mozilla compatibility.

Revision 3.2	2001−05−03

- Added Section 8.8, Introducing dialect.py .
- Added Section 7.2, Case Study: Street Addresses .
- Fixed bug in `handle_decl` method that would produce incorrect declarations (adding a space where it couldn't be).
- Fixed bug in CSS (introduced in 2.9) where body background color was missing.

Revision 3.1	2001−04−18

- Added code in `BaseHTMLProcessor.py` to handle declarations, now that Python 2.1 supports them.
- Added note about nested scopes in Section 8.5, locals and globals .
- Fixed obscure bug in Example 8.1, BaseHTMLProcessor.py where attribute values with character entities would not be properly escaped.
- Now recommending (but not requiring) Python 2.1, due to its support of declarations in `sgmllib.py`.
- Updated download links on the home page (http://diveintopython.org/) to point to Python 2.1, where available.
- Moved to versioned filenames, to help people who redistribute the book.

Revision 3.0	2001−04−16

- Fixed minor bug in code listing in Chapter 8, *HTML Processing*.
- Added link to Chinese translation on home page (http://diveintopython.org/).

Revision 2.9	2001−04−13

- Added Section 8.5, locals and globals .
- Added Section 8.6, Dictionary−based string formatting .
- Tightened code in Chapter 8, *HTML Processing*, specifically `ChefDialectizer`, to use fewer and simpler regular expressions.
- Fixed a stylesheet bug that was inserting blank pages between chapters in the PDF version.
- Fixed a script bug that was stripping the `DOCTYPE` from the home page (http://diveintopython.org/).
- Added link to Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/), and added a few links to individual recipes in Appendix A, *Further reading*.
- Switched to Google (http://www.google.com/services/free.html) for searching on `http://diveintopython.org/`.
- Upgraded to version 1.36 of the DocBook XSL stylesheets, which was much more difficult than it sounds. There may still be lingering bugs.

Revision 2.8	2001−03−26

- Added Section 8.3, Extracting data from HTML documents .
- Added Section 8.4, Introducing BaseHTMLProcessor.py .
- Added Section 8.7, Quoting attribute values .
- Tightened up code in Chapter 4, *The Power Of Introspection*, using the built−in function `callable` instead of manually checking types.
- Moved Section 5.2, Importing Modules Using from module import from Chapter 4, *The Power Of Introspection* to Chapter 5, *Objects and Object−Orientation*.
- Fixed typo in code example in Section 5.1, Diving In (added colon).
- Added several additional downloadable example scripts.
- Added Windows Help output format.

Revision 2.7	2001−03−16

* Added Section 8.2, Introducing sgmllib.py .
* Tightened up code in Chapter 8, *HTML Processing*.
* Changed code in Chapter 2, *Your First Python Program* to use `items` method instead of `keys`.
* Moved Section 3.4.2, Assigning Multiple Values at Once section to Chapter 2, *Your First Python Program*.
* Edited note about `join` string method, and provided a link to the new entry in *The Whole Python FAQ* (http://www.python.org/doc/FAQ.html) that explains why `join` is a string method (http://www.python.org/cgi−bin/faqw.py?query=4.96&querytype=simple&casefold=yes&req=search) instead of a list method.
* Rewrote Section 4.6, The Peculiar Nature of and and or to emphasize the fundamental nature of `and` and `or` and de−emphasize the `and−or` trick.
* Reorganized language comparisons into `note`s.

Revision 2.6	2001−02−28

* The PDF and Word versions now have colorized examples, an improved table of contents, and properly indented `tip`s and `note`s.
* The Word version is now in native Word format, compatible with Word 97.
* The PDF and text versions now have fewer problems with improperly converted special characters (like trademark symbols and curly quotes).
* Added link to download Word version for UNIX, in case some twisted soul wants to import it into StarOffice or something.
* Fixed several `note`s which were missing titles.
* Fixed stylesheets to work around bug in Internet Explorer 5 for Mac OS which caused colorized words in the examples to be displayed in the wrong font. (Hello?!? Microsoft? Which part of `<pre>` don't you understand?)
* Fixed archive corruption in Mac OS downloads.
* In first section of each chapter, added link to download examples. (My access logs show that people skim or skip the two pages where they could have downloaded them (the home page (http://diveintopython.org/) and preface), then scramble to find a download link once they actually start reading.)
* Tightened the home page (http://diveintopython.org/) and preface even more, in the hopes that someday someone will read them.
* Soon I hope to get back to actually writing this book instead of debugging it.

Revision 2.5	2001−02−23

* Added Section 6.4, Using sys.modules .
* Added Section 6.5, Working with Directories .
* Moved Example 6.17, Splitting Pathnames from Section 3.4.2, Assigning Multiple Values at Once to Section 6.5, Working with Directories .
* Added Section 6.6, Putting It All Together .
* Added Section 5.10, Summary .
* Added Section 8.1, Diving in .
* Fixed program listing in Example 6.10, Iterating Through a Dictionary which was missing a colon.

Revision 2.4.1	2001−02−12

* Changed newsgroup links to use "news:" protocol, now that `deja.com` is defunct.
* Added file sizes to download links.

Revision 2.4	2001−02−12

* Added "further reading" links in most sections, and collated them in Appendix A, *Further reading*.
* Added URLs in parentheses next to external links in text version.

Revision 2.3	2001−02−09

* Rewrote some of the code in Chapter 5, *Objects and Object−Orientation* to use class attributes and a better example of multi−variable assignment.
* Reorganized Chapter 5, *Objects and Object−Orientation* to put the class sections first.
* Added Section 5.8, Introducing Class Attributes .
* Added Section 6.1, Handling Exceptions .
* Added Section 6.2, Working with File Objects .
* Merged the "review" section in Chapter 5, *Objects and Object−Orientation* into Section 5.1, Diving In .
* Colorized all program listings and examples.
* Fixed important error in Section 2.2, Declaring Functions : functions that do not explicitly return a value return `None`, so you *can* assign the return value of such a function to a variable without raising an exception.
* Added minor clarifications to Section 2.3, Documenting Functions , Section 2.4, Everything Is an Object , and Section 3.4, Declaring variables .

Revision 2.2	2001−02−02

* Edited Section 4.4, Getting Object References With getattr .
* Added titles to `xref` tags, so they can have their cute little tooltips too.
* Changed the look of the revision history page.
* Fixed problem I introduced yesterday in my HTML post−processing script that was causing invalid HTML character references and breaking some browsers.
* Upgraded to version 1.29 of the DocBook XSL stylesheets.

Revision 2.1	2001−02−01

* Rewrote the example code of Chapter 4, *The Power Of Introspection* to use `getattr` instead of `exec` and `eval`, and rewrote explanatory text to match.
* Added example of list operators in Section 3.2, Introducing Lists .
* Added links to relevant sections in the summary lists at the end of each chapter (Section 3.8, Summary and Section 4.9, Summary).

Revision 2.0	2001−01−31

* Split Section 5.6, Special Class Methods into three sections, Section 5.5, Exploring UserDict: A Wrapper Class , Section 5.6, Special Class Methods , and Section 5.7, Advanced Special Class Methods .
* Changed notes on garbage collection to point out that Python 2.0 and later can handle circular references without additional coding.
* Fixed UNIX downloads to include all relevant files.

Revision 1.9	2001−01−15

* Removed introduction to Chapter 2, *Your First Python Program*.
* Removed introduction to Chapter 4, *The Power Of Introspection*.
* Removed introduction to Chapter 5, *Objects and Object−Orientation*.
* Edited text ruthlessly. I tend to ramble.

Revision 1.8	2001−01−12

* Added more examples to Section 3.4.2, Assigning Multiple Values at Once .
* Added Section 5.3, Defining Classes .
* Added Section 5.4, Instantiating Classes .

* Added Section 5.6, Special Class Methods .
* More minor stylesheet tweaks, including adding titles to `link` tags, which, if your browser is cool enough, will display a description of the link target in a cute little tooltip.

Revision 1.71	2001−01−03

* Made several modifications to stylesheets to improve browser compatibility.

Revision 1.7	2001−01−02

* Added introduction to Chapter 2, *Your First Python Program*.
* Added introduction to Chapter 4, *The Power Of Introspection*.
* Added review section to Chapter 5, *Objects and Object−Orientation* [later removed]
* Added Section 5.9, Private Functions .
* Added Section 6.3, Iterating with for Loops .
* Added Section 3.4.2, Assigning Multiple Values at Once .
* Wrote scripts to convert book to new output formats: one single HTML file, PDF, Microsoft Word 97, and plain text.
* Registered the `diveintopython.org` domain and moved the book there, along with links to download the book in all available output formats for offline reading.
* Modified the XSL stylesheets to change the header and footer navigation that displays on each page. The top of each page is branded with the domain name and book version, followed by a breadcrumb trail to jump back to the chapter table of contents, the main table of contents, or the site home page.

Revision 1.6	2000−12−11

* Added Section 4.8, Putting It All Together .
* Finished Chapter 4, *The Power Of Introspection* with Section 4.9, Summary .
* Started Chapter 5, *Objects and Object−Orientation* with Section 5.1, Diving In .

Revision 1.5	2000−11−22

* Added Section 4.6, The Peculiar Nature of and and or .
* Added Section 4.7, Using lambda Functions .
* Added appendix that lists section abstracts.
* Added appendix that lists tips.
* Added appendix that lists examples.
* Added appendix that lists revision history.
* Expanded example of mapping lists in Section 3.6, Mapping Lists .
* Encapsulated several more common phrases into entities.
* Upgraded to version 1.25 of the DocBook XSL stylesheets.

Revision 1.4	2000−11−14

* Added Section 4.5, Filtering Lists .
* Added `dir` documentation to Section 4.3, Using type, str, dir, and Other Built−In Functions .
* Added `in` example in Section 3.3, Introducing Tuples .
* Added additional note about `if __name__` trick under MacPython.
* Switched to the SAXON XSLT processor from Michael Kay.
* Upgraded to version 1.24 of the DocBook XSL stylesheets.
* Added db−html processing instructions with explicit filenames of each chapter and section, to allow deep links to content even if I add or re−arrange sections later.
* Made several common phrases into entities for easier reuse.
* Changed several `literal` tags to `constant`.

Revision 1.3	2000−11−09

• Added section on dynamic code execution. • Added links to relevant section/example wherever I refer to previously covered concepts. • Expanded introduction of chapter 2 to explain what the function actually does. • Explicitly placed example code under the GNU General Public License and added appendix to display license. [Note 8/16/2001: code has been re−licensed under GPL−compatible Python license] • Changed links to licenses to use `xref` tags, now that I know how to use them.	
Revision 1.2	2000−11−06
• Added first four sections of chapter 2. • Tightened up preface even more, and added link to Mac OS version of Python. • Filled out examples in "Mapping lists" and "Joining strings" to show logical progression. • Added output in chapter 1 summary.	
Revision 1.1	2000−10−31
• Finished chapter 1 with sections on mapping and joining, and a chapter summary. • Toned down the preface, added links to introductions for non−programmers. • Fixed several typos.	
Revision 1.0	2000−10−30
• Initial publication	

Appendix F. About the book

This book was written in DocBook XML (http://www.oasis−open.org/docbook/) using Emacs (http://www.gnu.org/software/emacs/), and converted to HTML using the SAXON XSLT processor from Michael Kay (http://saxon.sourceforge.net/) with a customized version of Norman Walsh's XSL stylesheets (http://www.nwalsh.com/xsl/). From there, it was converted to PDF using HTMLDoc (http://www.easysw.com/htmldoc/), and to plain text using w3m (http://ei5nazha.yz.yamagata−u.ac.jp/~aito/w3m/eng/). Program listings and examples were colorized using an updated version of Just van Rossum's `pyfontify.py`, which is included in the example scripts.

If you're interested in learning more about DocBook for technical writing, you can download the XML source (http://diveintopython.org/download/diveintopython−xml−5.4.zip) and the build scripts (http://diveintopython.org/download/diveintopython−common−5.4.zip), which include the customized XSL stylesheets used to create all the different formats of the book. You should also read the canonical book, *DocBook: The Definitive Guide* (http://www.docbook.org/). If you're going to do any serious writing in DocBook, I would recommend subscribing to the DocBook mailing lists (http://lists.oasis−open.org/archives/).

Appendix G. GNU Free Documentation License

Version 1.1, March 2000

> Copyright (C) 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

G.0. Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

G.1. Applicability and definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front−matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front−Cover Texts or Back−Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine−readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats

suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard−conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine−generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

G.2. Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

G.3. Copying in quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front−Cover Texts on the front cover, and Back−Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine−readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly−accessible computer−network location containing a complete Transparent copy of the Document, free of added material, which the general network−using public has access to download anonymously at no charge using public−standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

G.4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front−matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties−−for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front−Cover Text, and a passage of up to 25 words as a Back−Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front−Cover Text and one of Back−Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the

previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

G.5. Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

G.6. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

G.7. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self−contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

G.8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

G.9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

G.10. Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/ (http://www.gnu.org/copyleft/).

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

G.11. How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front−Cover Texts being LIST, and with the Back−Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front−Cover Texts, write "no Front−Cover Texts" instead of "Front−Cover Texts being LIST"; likewise for Back−Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix H. Python license

H.A. History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI) in the Netherlands as a successor of a language called ABC. Guido is Python's principal author, although it includes many contributions from others. The last version released from CWI was Python 1.2. In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI) in Reston, Virginia where he released several versions of the software. Python 1.6 was the last of the versions released by CNRI. In 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. Python 2.0 was the first and only release from BeOpen.com.

Following the release of Python 1.6, and after Guido van Rossum left CNRI to work with commercial software developers, it became clear that the ability to use Python with software available under the GNU Public License (GPL) was very desirable. CNRI and the Free Software Foundation (FSF) interacted to develop enabling wording changes to the Python license. Python 1.6.1 is essentially the same as Python 1.6, with a few minor bug fixes, and with a different license that enables later versions to be GPL−compatible. Python 2.1 is a derivative work of Python 1.6.1, as well as of Python 2.0.

After Python 2.0 was released by BeOpen.com, Guido van Rossum and the other PythonLabs developers joined Digital Creations. All intellectual property added from this point on, starting with Python 2.1 and its alpha and beta releases, is owned by the Python Software Foundation (PSF), a non−profit modeled after the Apache Software Foundation. See http://www.python.org/psf/ for more information about the PSF.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

H.B. Terms and conditions for accessing or otherwise using Python

H.B.1. PSF license agreement

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.1.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty−free, world−wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.1.1 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright (c) 2001 Python Software Foundation; All Rights Reserved" are retained in Python 2.1.1 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.1.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.1.1.
4. PSF is making Python 2.1.1 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.1.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.1.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.1.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.1.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.2. BeOpen Python open source license agreement version 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non−exclusive, royalty−free, world−wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.3. CNRI open source GPL−compatible license agreement

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty−free, world−wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright (c) 1995−2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement

may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non−separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.4. CWI permissions statement and disclaimer

> Copyright (c) 1991 − 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.